



DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed eliminaremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



CoScienze
Associazione

Luca Morelli

IGES



CoScienze
Associazione

Indice	i
--------	---

Elenco delle figure	xix
---------------------	-----

Elenco delle tabelle	xxiii
----------------------	-------

1 Introduzione	1
-----------------------	---

1.1 Concetti base	1
-----------------------------	---

1.2 Leggi di Lehman	2
-------------------------------	---

1.2.1 Continuing change	2
-----------------------------------	---

1.2.2 Increasing complexity	2
---------------------------------------	---

1.2.3 Leggi di Lehman: conclusioni	2
--	---

1.3 Software maintenance	3
------------------------------------	---

1.4 Software maintenance secondo Kitchenam	4
--	---

1.5 Standard di manutenzione	5
--	---

1.6 Software configuration management	6
---	---

1.7 Reengineering	6
-----------------------------	---

1.8 Legacy Sistems	7
------------------------------	---

1.8.1 Strategie dei legacy systems	7
--	---

1.9 Impact analysis	7
-------------------------------	---

1.10 Refactoring	8
----------------------------	---

1.11 Program comprehension	8
--------------------------------------	---

1.12 Software reuse	8
-------------------------------	---

2 Processo di manutenzione	9
2.1 Introduzione: processo di manutenzione	9
2.2 Prodotto	9
2.3 Tipi di manutenzione	10
2.4 Processi della maintenance organization	10
2.4.1 Livelli di help desk	11
2.5 Peopleware	12
2.6 User & Customer issues	12
3 Evoluzione del software	13
3.1 Introduzione: Evoluzione del software	13
3.2 Approcci alla software evolution	13
3.3 SPE Taxonomy	14
3.4 Leggi della software evolution	15
3.5 Aging	17
3.6 Code decay	18
3.7 COTS based systems	18
3.7.1 Maintenance dei CBS	19
4 Ciclo di vita del software	20
4.1 Introduzione: ciclo di vita del software	20
4.2 Cicli di vita per la manutenzione	21
4.3 Sviluppo iterativo e sviluppo incrementale	23
4.4 Gli staged models di CSS	24
4.5 Staged model per FLOSS	25
4.6 Change Mini-Cycle Model	25
4.7 Gli standard della software maintenance	26
4.7.1 IEEE/EIA 1219	26
4.7.2 ISO/IEC 14764	29
5 Ciclo di vita del software	33
5.1 Introduzione: Software Configuration Management	33
5.2 SCM Spectrum of Functionality	34
5.3 Product	35
5.3.1 Identification	35

5.3.2	Version control	35
5.3.3	System models and selections	36
5.4	Tool	37
5.4.1	Workspace control	37
5.4.2	Building	38
5.5	Process	38
5.5.1	change management	38
5.5.2	Accounting	38
5.5.3	Auditing	39
5.6	SCM Process	39
5.7	Change request flow	40
5.8	Baseline	41
6	Reengineering	42
6.1	Introduzione: Reengineering	42
6.2	Concetti del reengineering	42
6.2.1	Astrazione	43
6.2.2	Raffinamento	43
6.2.3	Forward e reverse engineering	43
6.2.4	Livelli di astrazione	44
6.3	Principio di alterazione	44
6.4	Rehosting	45
6.5	Tipi di cambiamenti del reengineering	45
6.6	Rewrite	46
6.7	Rework	46
6.8	Replace	47
6.9	Reengineering variations	47
6.10	Reengineering Process	48
7	Code reverse engineering	50
7.1	Introduzione: Code reverse engineering	50
7.2	Definizione formale di code reverse engineering	51
7.3	Design recovery e Code reverse engineering	51
7.4	Obiettivi del reverse engineering	52
7.5	Standard e reverse engineering	53

7.6	Esecuzione simbolica	54
7.7	Reverse enginerring ed altri utilizzi	54
7.8	Paradigma Goal/Model/Tool	55
7.8.1	Goals	55
7.8.2	Models	55
7.8.3	Tools	56
7.9	Tecniche di facilitazione	56
7.10	Decompilation e Compilation	59
7.11	Data reverse engineering	59
7.11.1	Fai del data reverse engineering	59
7.12	Analisi statica nella compilazione	60
8	Grafi e modelli di rappresentazione dei programmi	62
8.1	Grafi	62
8.2	Cammini e cicli	62
8.3	Raggiungibilità di grafi non orientati	62
8.4	Control flow graph	63
8.5	GFC ben formati	63
8.6	Costruzione del control flow graph	64
8.7	Analisi del flusso dati	64
8.8	Regole del CFG	65
8.9	Espressioni cammino/variabile	66
8.10	Espressioni regolari	66
8.11	Algoritmi definizione/uso	66
8.12	Program slicing	67
8.13	Slice	67
8.14	Traiettoria di stato	68
8.15	Criterio di slicing	68
8.16	Definizione formale di slice	69
8.17	Calcolo della slice minima	69
8.18	Slicing e Program Dependence Graph	70
8.19	Dipendenze sul controllo	71
8.20	Control Dependence Graph	72
8.21	Dipendenze sui dati	72

8.22 Program dependence graph	73
8.22.1 Algoritmo di slicing basato su PDG	73
8.23 Backward vs forward slicing	73
8.24 Decomposition slicing	74
8.25 Slicing interprocedurale	74
8.26 System dependence graph	75
8.27 Usi dello slicing	76
8.28 Client-Server restructuring	76
8.29 Individuazione del criterio di slicing	76
8.30 Slicing dinamico	77
8.31 Quasi static slicing	78
8.32 Conditioned slicing	78
8.33 Amorphous program slicing	79
8.34 Problema dello slicing	79
9 Impact analysis	80
9.1 Introduzione: Impact analysis	80
9.2 Processo di impact analysis	81
9.3 Tipi di tracciabilità	81
9.4 Metriche dell'information retrieval	82
9.5 Adequacy	83
9.6 Effectiveness	83
9.7 Identificare il SIS	84
9.8 Analisi del traceability graph	84
9.9 Identificare il CIS	85
9.10 Risoluzione della changeRate	85
9.11 Call graph	86
9.12 Program dependence graph	86
9.13 Ripple effect	87
10 Testing	88
10.1 Verifica e convalida	88
10.2 Dependability	89
10.3 Verifica statica e dinamica	89
10.4 Testing and debugging	90

10.5 Definizioni di testing	91
10.6 Concetti per testing & debugging	91
10.7 Test e casi di test	92
10.8 Oracolo	92
10.9 Problemi del testing	92
10.9.1 Problemi del testing: Equivalenza di funzioni	92
10.10 Validazione e verifica: confidenza	93
10.11 Terminazione del testing	93
10.12 Test ideale	93
10.13 Criterio di selezione di test	94
10.14 Tecniche di testing	94
10.15 Random testing & testing sistematico	95
10.16 Principio di partizionamento	95
10.17 Analisi mutazionale	96
10.18 Testing statistico e affidabilità del software	97
11 Pianificazione del testing	98
11.1 Introduzione: pianificazione del testing	98
11.2 Modello di sviluppo a V	98
11.3 Livelli di testing	99
11.4 Testing di unità	99
11.5 Problema dello scaffolding	100
11.6 Generazione test driver e test stub	100
11.6.1 Software OO e Unit Testing	100
11.6.2 Information hiding	101
11.7 Testing di integrazione	101
11.7.1 Modalità di testing di integrazione	102
11.7.2 Top down	103
11.7.3 Bottom up	103
11.7.4 Sandwitch	103
11.7.5 Grafo delle dipendenze	104
11.7.6 Problemi di integrazione per SOO	105
11.8 Testing di sistema	105
11.9 Test di accettazione	105

11.10 Alpha e beta testing	105
12 Documenti di testing	107
12.1 Introduzione: documenti di pianificazione	107
12.2 Documenti di pianificazione e specifica	107
12.2.1 Master Test Plan (MTP)	107
12.2.2 Level test plan	108
12.2.3 Level test deisng	108
12.2.4 Level test case	109
12.2.5 Level test procedure	110
12.3 Documenti di esecuzione	110
12.3.1 Level Test Log (TL)	110
12.3.2 Anomaly report (TIR)	110
12.4 Level Interim Test Status Report (LITSR)	111
12.5 Level Test Report (LTR)	111
12.6 Master Test Report (MTR)	112
13 Testing funzionale o testing black box	113
13.1 Introduzione: Testing funzionale	113
13.2 Black box & white box: differenze	113
13.3 Scalabilità del black box	114
13.4 Riuso black box & white box	114
13.5 Tecniche di testing funzionale	115
13.6 Criteri di copertura per testing funzionale	115
13.7 Suddivisione in classi di equivalenza	115
13.8 Criteri di copertura deboli e forti	116
13.9 Selezione delle classi di equivalenza	117
13.10 Testing dei valori limite (Boundary values)	118
13.11 Worst case testing	119
13.12 Metodi per ridurre i casi di prova	120
13.13 Category partition	120
13.13.1 Vincoli	121
13.13.2 Proprietà delle scelte	122
13.13.3 Steps del category partition	122
13.13.4 Conclusioni	123

13.14 Tabelle di decisioni	124
13.14.1 Condizioni di uso ideali	125
13.14.2 Scalabilità	125
13.14.3 Condizioni speciali	125
13.15 Metodi dei grafi causa effetto	126
13.15.1 Struttura del grafo causa effetto	126
13.15.2 Derivazione dei casi di test	127
13.16 Pairwise or n-way combinatorial testing	127
13.16.1 Pairwise combination (invece che esaustiva)	127
14 Testing strutturale o testing white box	128
14.1 Introduzione: testing white box o testing strutturale	128
14.2 Selezionare i casi di test con tecniche white box	129
14.3 Statement coverage	129
14.4 Copertura delle decisioni o branch coverage	129
14.5 Copertura delle condizioni	130
14.6 Copertura di decisioni e condizioni	130
14.7 Copertura dei cammini o path coverage	131
14.8 Criterio di n-copertura dei cicli	131
14.9 Metodo degli exemplar path	132
14.10 Partizionamento perfetto e quasi-perfetto	132
14.11 Esecuzione simbolica	132
14.11.1 Path condition	133
14.11.2 Feasible & unfeasible path	133
14.12 Execution three	133
14.13 Problemi di raggiungibilità indecidibili	134
14.14 Metodo di McCabe	134
14.14.1 Funzionamento del metodo di McCabe	134
14.14.2 Insieme dei cammini base	135
14.14.3 Criterio di McCabe	135
14.14.4 Teoria ciclomatica dei grafi e McCabe	136
14.14.5 Calcolare il numero ciclomatico di McCabe	136
14.14.6 Ricerca dei percorsi indipendenti	137
14.15 Metodi fondati su data flow	138

14.15.1 Tipi delle variabili	138
14.16 Tipi di path	138
14.17 Criteri strutturali e livelli di testing	140
14.18 GCOV	140
14.19 Software testing	140
14.20 Stato ed information hiding	140
14.21 Macchine a stati finiti e generazione di casi di test	141
14.21.1 Macchina a stati gerarchica	141
14.21.2 Criterio di copertura	142
15 Testing di integrazione	143
15.1 Testing di integrazione basato su branch	143
15.2 Problemi del testing di integrazione	143
15.3 Ereditarietà	144
15.3.1 Possibili soluzioni all'ereditarietà	144
15.4 Polimorfismo	144
15.4.1 Possibili soluzioni del polimorfismo	145
15.5 Gestione delle eccezioni	145
15.6 Concorrenza	146
16 Ispezione	147
16.1 Introduzione: Ispezione	147
16.2 Fasi di ispezione	148
16.3 Incentivi	148
16.4 Varianti	149
16.5 Esempio: Checklist in Java ed in C	149
17 Standard IOS/IEC 12207	151
17.1 Introduzione: ISO/IEC 12207	151
17.2 Principi di base	152
17.3 Architettura e processi	152
17.4 Classi di processi	152
17.5 Processo di agreement	153
17.5.1 Acquisition process	153
17.6 Supply process	154

17.7 Organizational Project-Enabling Processes	154
17.8 Altri tipi processi	155
17.9 Software Reuse Processes	156
18 Project management	157
18.1 Introduzione: Project management	157
18.2 Software project management	158
18.3 Management activities	158
18.4 Progetto software	159
18.5 Software Project Management Plan (SPMP)	160
18.6 Componenti di un progetto	160
18.7 Stati di un progetto	161
18.8 Aspetti di un progetto	161
18.9 Project agreement	162
18.10 IEEE 1058	162
18.11 Project agreement, problem statement e SPMP	162
18.12 Organizzazione di un SPMP	163
18.13 Risk management	164
18.13.1 Fattori di rischio nel software project	164
18.13.2 Processo di risk management	165
18.14 Work package e work product	167
18.15 Work Breakdown Structure (WBS)	167
18.15.1 Da WBS a Dependency graph	167
18.16 PERT Chart	168
18.17 Dependency diagram	168
18.18 Activity diagram	169
18.19 Action item	170
18.20 Conclusioni: Leggi di progetto	170
18.21 Problemi di scheduling	170
19 Legacy system	172
19.1 Introduzione: legacy systems	172
19.2 LIS e decomponibilità	172
19.3 Soluzione per i Legacy Information Systems (LIS)	173
19.4 Wrapping	173

19.5 Tipi di wrapping	174
19.5.1 Database wrappers	174
19.5.2 System service wrapper	174
19.5.3 Application wrapper	175
19.5.4 Function wrapper	175
19.6 Livelli di incapsulamento	175
19.6.1 Process level	176
19.6.2 Transaction level	176
19.6.3 Program level	176
19.6.4 Module level	176
19.6.5 Procedure level	177
19.7 Costruire un wrapper	177
19.7.1 External interface	178
19.7.2 Internal interface	178
19.7.3 Message handler	179
19.7.4 Interface converter	179
19.7.5 I/O Emulator	179
19.8 Adattare un programma al wrapper	179
19.9 Screen scraping	179
19.10 Migration	180
19.11 Schema conversion	181
19.12 Data conversion	181
19.13 Program conversion	181
19.14 Coerenza di LIS e sistema target	182
19.15 Cut over e Roll over	182
19.16 Migration planning	182
19.16.1 Perform portfolio analysis	183
19.16.2 Identificare gli stakeholders	184
19.16.3 Creare un business case	184
19.16.4 Go-on o no-go decisions	184
19.16.5 Comprendere il LIS	185
19.16.6 Comprendere la target technology	185
19.16.7 Definire la target architecture	185
19.16.8 Definire la strategia	185

19.16.9 Riconciliare la tecnologia con le necessità degli stackholders	186
19.16.10 Determinare le risorse necessarie	186
19.16.11 Valutare la fattibilità della strategia	186
19.17 Metodi di migrazione	186
19.17.1 Cold turkey	187
19.17.2 Database first	187
19.17.3 Database last	188
19.17.4 Composite database	188
19.17.5 Chicken little	189
19.17.6 Butterfly	190
20 Metriche del software	191
20.1 Introduzione: Misure ed ingegneria del software	191
20.2 Modelli di qualità	192
20.3 Teoria della rappresentazione della misura	192
20.4 Misure	193
20.5 Misure dirette ed indirette	193
20.6 Sistema di relazioni empiriche	193
20.7 Sistema di relazioni numeriche	194
20.8 Relazioni tra sistemi di relazione	194
20.9 Scala	194
20.10 Trasformazioni di scala ammissibili	195
20.11 Tipi di scale	195
20.12 Sensatezza di affermazione	196
20.13 Tipi di scale ed operazioni ammesse	196
20.14 Misure indirette e trasformazioni di scala ammissibili	197
20.15 Misura del software	197
20.16 Attributi interni ed esterni	197
20.17 Misure: valutazione e stima	198
20.18 Sistemi predittivi	198
20.19 Sistemi predittivi e misure	198
20.20 Comprendere la realtà e poi misurarla	199
20.21 Progettazione degli esperimenti ed analisi dei risultati	199
20.22 Esperimenti di misura	199

20.23 Strategie di sperimentazione	200
20.24 Strategie di accoppiamento	200
20.25 Analisi dei risultati	201
20.26 Validazione delle misure del software	202
20.26.1 Validazione di un sistema predittivo	202
20.26.2 Validazione di una misura	202
20.26.3 Validazione di una misura: errori	202
20.27 Caso del numero ciclomatico di McCabe	203
20.28 Raccolta ed analisi dei dati	203
20.28.1 Come raccogliere i dati	203
20.28.2 Quando raccogliere i dati	204
20.28.3 Come conservare i dati	204
20.28.4 Come analizzare i dati	204
20.29 Statistiche	204
20.29.1 Box Plot	204
20.30 Relazione tra due attributi	205
20.30.1 Regressione lineare	205
20.31 Regressione logaritmica	206
20.32 Parametri calcolati in Excel	206
20.33 Misure in ambito industriale	207
21 Software quality management	208
21.1 Introduzione: software quality management	208
21.2 Qualità	208
21.3 Attività di quality management	209
21.4 ISO 9000	209
21.4.1 Certificazione ISO 9000	210
21.4.2 ISO 9000 e quality management	210
21.5 Importanza degli standard	210
21.6 Problemi con gli standard	210
21.7 Standard di documentazione	211
21.8 Qualità del processo e qualità del prodotto	211
21.9 Process based quality	211
21.10 Practical process quality	212

21.11 Quality planning	213
21.11.1 Quality plan structure	213
21.12 Quality control	213
21.13 Quality review	213
21.13.1 Review functions	214
21.14 Software measurement and metrics	214
21.15 Processo di misurazione	215
21.16 Data collection	215
21.17 Metriche statiche e dinamiche	216
21.18 Measurement analysis	216
22 Software cost estimation	217
22.1 Introduzione: Software cost estimation	217
22.2 Componenti di costo	218
22.3 Costing and pricing	218
22.4 Produttività del programma	219
22.5 Misure di produttività	219
22.6 Stima della size di un software	219
22.7 Linee di codice	220
22.8 Function points	221
22.9 Object points	221
22.10 Fattori che influenzano la produttività	222
22.11 Tecniche di stima dei costi	222
22.12 Tecniche di stima dei costi: Problemi	223
22.13 Stima top down e bottom up	223
22.13.1 Stima top down	224
22.13.2 Stima bottom up	224
22.14 Work Breakdown Structure	224
22.15 Metodi di stima	224
22.16 Pricing to win	225
22.17 Stima per analogia	225
22.18 Stima dettagliata	226
22.19 Modelli algoritmici	227
22.20 Basic COCOMO: Organic mode	228

22.20.1 Limiti di Basic COCOMO	228
22.21 Diseconomia di scala	228
22.22 Intermediate COCOMO	229
22.23 Intermediate COCOMO: cost drivers	230
22.24 Problemi dello sviluppo software	230
22.25 Attività di sviluppo	230
22.26 Maturità del processo	231
22.27 Livelli del CMM	231
22.28 Modello a cascata	232
22.29 Modello a V	232
22.30 Modello a spirale	232
22.31 Process improvement	233
22.32 Attributi di processo	233
22.33 Stati della process improvement	234
22.34 Qualità del processo e del prodotto	235
22.35 Analisi e modellazione	235
22.36 Activity diagram con object flow	236
22.37 Eccezioni di un processo	236
22.38 Process measurement	237
22.39 Paradigma Goal/Questions/Metrics	237
22.40 Software Engineering Institute	237
23 Modelli agili	240
23.1 Scrum	240
23.1.1 Sprints	240
23.2 Scrum's characteristics	241
23.3 Agile Manifesto	242
23.4 Scrum framework	243
23.5 Scalability	245
23.6 Capacità del team e Focus Factor	245
23.7 Stimare le user stories	246
23.8 Planning Poker	246
23.9 eXtreme Programming	246
23.10 Qualità e testing	247

23.11 Quattro valori dell'eXtreme Programming	247
23.12 Pair programming	248
23.13 Cinque principi di base dell'eXtreme Programming	248
23.14 Le 12 pratiche dell'eXtreme Programming	249
23.15 Problemi dell'eXtreme Programming	250
23.16 Kanban	250
23.17 Funzionamento di Kanban	251
23.18 Policy e Postit di Kanban	251
23.19 Termini importanti di Kanban	251
23.20 Limitare il WiP	252
23.21 Ottenerne ed incoraggiare il feedback	252
23.21.1 Kanban Meeting	252
23.21.2 Replenishment Meeting	252
23.22 Ridurre i rischi di ritardi	253
23.23 Differenza tra Scrum e Kanban	253
23.24 DevOps	254
23.25 Flusso di valore	254
23.26 Lead time	254
23.27 Problemi di DevOps	255
23.28 Risoluzione del problema di DevOps	255
23.29 Fasi di DevOps	255
23.30 Garantire DevOps	256
23.31 Categorie di tecniche del DevOps	256
23.32 Continous architecting	257
23.33 Continous integration	257
23.34 Continous testing	258
24 Code smells	260
24.1 Introduzione: Problemi sul codice	260
24.2 Code smell	260
24.3 Refactoring	262
24.3.1 Extract class refactoring	262
24.3.2 Move method refactoring	263
24.4 Principi che guidano il refactoring	263

24.5 Introduzione di code smells nella pratica	263
24.6 Percezione dei code smells	263
24.7 Refactoring process	264
24.8 Where to refactor	264
24.9 DECOR	264
24.10HIST	265
24.11Lessico del codice e code smells	265
24.12How to refactor	266
24.13Supporto automatico	267
24.14Tipi di relazioni nei code smell	267
24.15Scelta dell'algoritmo per generare la soluzione	268
24.16Guarantee behaviour preservation	268
24.17Apply the refactoring & Access its effects on quality	268
24.18Consistently modify other artifacts	269
24.19Effetti collaterali del refactoring	269
24.20Impatto sulla qualità	269
24.21Graph trasformation	270
24.22ARIES	270
24.23System mutation	271
25 Test di regressione	272
25.1 Introduzione: test e refactoring	272
25.2 Testing di regressione	272
25.3 Test all	273
25.4 Selective testing	273
25.5 Tecniche di ottimizzazione	273
25.6 Test suite minimization	273
25.7 Variante della test suite minimization	274
25.7.1 Minimization: algoritmo greedy	274
25.7.2 Minimization: additional greedy	274
25.8 Regression test selection	275
25.8.1 Test case selection	275
25.8.2 Weighted Set Cover	276
25.8.3 Selection: algoritmo greedy	276

25.8.4 Selection: additional greedy	276
25.9 Test case prioritization	277
25.9.1 Funzione APFD	277
25.9.2 Problemi dell'APFD e soluzioni	278
25.9.3 Definizione formale di test case prioritization	278
25.9.4 Prioritization: Algoritmo greedy	278
25.10 Conclusioni: testing di ottimizzazione	279
25.11 Algoritmi genetici e software engineering	279
25.12 Backtracking & Branch and bound	279
25.13 Exploration ed exploitation	280
25.14 Hill climbing	280
25.15 Problemi di ottimizzazione	281
25.16 SBSE	281
25.17 Algoritmi genetici e SE	282
25.18 No free lunch teoreme (NFL)	282
25.19 Fasi degli algoritmi genetici	283
25.20 GAs, exploration & exploitation	284
25.21 Tipi di selezione	284
25.22 Automated input generation	285
25.23 Test data generation: Fitness function	285
25.24 Collateral coverage	287
25.25 Problemi multi obiettivo (MOP)	287
25.26 Concetto di dominanza	288
25.27 Fronte di Pareto	288
25.28 GAs e MOPs	289
25.29 Non-Dominated Sorting Algorithm (NSGA)	289
25.30 Fast NSGA	290
25.31 GAs e testing di regressione	290
25.31.1 Test suite minimization & test case selection	290
25.32 Test suite prioritization	290
25.33 Ipervolume	291
25.34 GAs: conclusioni	291
25.35 Single value decomposition	292

Elenco delle figure

2.1 Processo di manutenzione	9
3.1 S type	14
3.2 P type	15
3.3 E type	15
4.1 ciclo di vita del software	20
4.2 Fix model	21
4.3 Iterative model	22
4.4 Full reuse model	22
4.5 Change Mini-Cycle Model	25
5.1 SCM functionalities	35
5.2 Version control	36
5.3 SCM process	40
5.4 Change request flow	40
6.1 Livelli di astrazione	44
6.2 Principio di alterazione	44
6.3 Modello a ferro di cavallo	45
6.4 Rewrite, rework e replace	47
6.5 Rewrite, rework e replace	47
7.1 Code reverse engineering	50

7.2 Code reverse engineering	52
8.1 Conditioned slicing	79
9.1 Analisi del traceability graph	85
9.2 Call graph	86
9.3 Call graph	87
10.1 Verifica e convalida	89
11.1 Verifica e convalida	99
11.2 Verifica e convalida	104
11.3 Verifica e convalida	104
12.1 Verifica e convalida	108
12.2 Verifica e convalida	112
13.1 Boundary analysis	119
13.2 WCT	119
13.3 Vincoli	121
13.4 Tabella di decisioni	124
13.5 Tabella di decisioni: Esempio	125
13.6 Grafo cause effetto	127
14.1 Numero ciclomatico di McCabe	136
14.2 Percorsi indipendenti	137
14.3 Percorsi indipendenti	137
14.4 Percorsi indipendenti	137
14.5 Tipi di path	139
15.1 Polimorfismo	145
17.1 Classi di processi	153
18.1 Componenti di un progetto	160
18.2 Stati di un progetto	161
18.3 Project agreement, problem statement e SPMP	163
18.4 Risk management steps	167
18.5 PERT Chart	168

18.6 Dependency diagram	169
19.1 Database Wrapper	175
19.2 Livelli di encapsulamento	176
19.3 Costruire un wrapper	177
19.4 Perform portfolio analysis	184
19.5 Database first	187
19.6 Database last	188
19.7 Composite database	189
19.8 Chicken little	189
20.1 Tipi di scale	196
20.2 Attributi interni ed esterni	198
20.3 Box Plot	205
20.4 Box Plot	205
20.5 Scatterplot	206
21.1 Process based quality	212
21.2 Software measurement and metrics	215
21.3 Data collection	216
22.1 Problemi di misura	220
22.2 Stima per analogia	226
22.3 Basic COCOMO: Organic mode	228
22.4 Diseconomia di scala	229
22.5 Modello a V	232
22.6 Stati della process improvement	235
22.7 Software Engineering Institute	239
23.1 Scrum process	241
23.2 standard normalization process	242
23.3 Burndown chart	245
23.4 eXtreme Programming	247
23.5 Passaggi di DevOps	255
25.1 algoritmo greedy	275
25.2 additional greedy	275

25.3 Selection: algoritmo greedy	276
25.4 Selection: additional greedy	277
25.5 Prioritization: Algoritmo greedy	279
25.6 Hill climbing	281
25.7 Algoritmi genetici	284
25.8 Approach level	286
25.9 Branch distance	286
25.10Branch distance	287
25.11NSGA	290
25.12Test suite prioritization	291



Elenco delle tabelle



CAPITOLO 1

Introduzione

1.1 Concetti base



Con **maintenance** parliamo di un processo, ovvero **attività che vengono svolte da sviluppatori per modificare un sistema**, mentre con **evolution** parliamo di ciò che accade al sistema come conseguenza della manutenzione. Il cambiamento del software comporta **un'evoluzione del software**. Per capire come si evolve il software nel tempo senza però sapere gli interventi di manutenzioni e le richieste di manutenzione, è necessario **guardare le varie versioni del software** e cosa è cambiato tra esse. Si può studiare l'evoluzione di un sistema anche senza sapere gli interventi di manutenzione o senza partecipare ad essi. Esiste una branca dell'ingegneria del software che si chiamata **mining software repositories** che applica tecniche di data mining alle repository software per studiare e capire gli eventi che hanno impattato il sistema software.

L'evoluzione può essere sì vista se ci sono stati cambiamenti ai requisiti, ma il modo corretto di vedere evolution è da **definire dal punto di vista software**. Mentre i processi sono **le attività che devono essere fatte per produrre il software**, quando si parla del **ciclo di vita del software**, si parla di **ciò che accade nella vita di un software**.

Usando determinati modelli di processo, come quello a cascata, anche il ciclo di vita del software seguirà lo stesso modello, ovvero quello a cascata. I due concetti sono però differenti. L'**evoluzione guarda alle attività svolte dagli sviluppatori**, mentre il **ciclo di vita del software guarda ciò che succede al software**. maintenance quindi è un processo mentre

evolution fa parte del ciclo di vita del software. Si usa il concetto di evolution per definire la crescita dinamica del software.

1.2 Leggi di Lehman

Alcune osservazioni sono indipendenti dal tipo di software sviluppato, e quindi valgono per tutte, mentre altre sono afferenti del contesto in cui sono state definite. Esse afferiscono agli evolving systems, ovvero sistemi che influenzano la realtà e la realtà influenza il sistema. Le osservazioni **indipendenti dal contesto** sono:

1.2.1 Continuing change

Un sistema diventerà progressivamente meno soddisfacente per i suoi utenti, a meno che non sia continuamente adattata per incontrare i nuovi bisogni. Il **sistema è costretto a cambiare**, in quanto la realtà in cui opera cambia. Nel caso in cui il sistema non cambi, il sistema perde di utilità dovendo venire abbandonato o rimpiazzato da un altro sistema.

1.2.2 Increasing complexity

Poiché un sistema cambia, la sua complessità aumenta, per cui decresce la qualità. Le modifiche sostanziali al sistema possono introdurre difetti, che è necessario poi risolvere. C'è quindi un declino costante della qualità software ed un conseguente aumento della complessità. Più la complessità aumenta più è complesso apportare nuove modifiche, ecco perché la qualità decresce, in quanto sarà più difficile modificare senza ridurre la qualità del sistema. L'unico modo per **ridurre la complessità è tramite reengineering**, migliorando quindi la qualità del sistema software mediante operazioni mirate.

1.2.3 Leggi di Lehman: conclusioni

Migliorare la qualità del codice senza nessuna aggiunta non porta quasi mai a guadagni. Per poter convincere i project manager in un intervento di reengineering è **necessario mostrare come gli interventi fatti abbiano introdotto debiti tecnici**, che costeranno di più in fase di manutenzione se non viene fatta nessuna operazione mirata per eliminare i debiti tecnici sul momento. Queste leggi, **sviluppate per i sistemi ClosedSource, non valgono**, almeno in parte, **per i sistemi OpenSource** (escluse le prime due che valgono per qualsiasi sistema). Ciò è ovvio perché, nel caso di OpenSource, si sta parlando di sistemi che vanno in rete e che

non rimangono in un mainframe. I sistemi OpenSource possono inoltre essere organizzati diversamente. In sistemi tali si potrebbe avere una maggior crescita dei contributors.

1.3 Software maintenance

Inizialmente visto per correggere i difetti, come ad esempio gold fix, o waterfall model. Il modello successivo è stato l'iterative manteinance. Si è poi arrivati agli standard sulla manutenzione, per cui lo standard IEEE 12129 definisce 3 tipi di manutenzioni: Lo **standard IEEE 1219** definisce 3 tipi di manutenzioni:

- **correttiva:** per correggere failures, che possono essere failure di performance o elaborazione. A causa di interventi di manutenzione si possono avere failures funzionali, ovvero il sistema non funziona più come prima, ma si possono avere anche failures di performance, ovvero a causa della modifica il sistema non ha più la stessa efficienza richiesta, per cui bisogna intervenire. Il processo di corrective manteinance è simile a quello di debugging. Si va ad individuare il fault che causa la failure ed eliminarlo, anche se il debugging ha a che fare solo con il codice, mentre il processo di corrective maintenance afferisce a tutte le fasi del ciclo di vita del software. Alla fine però l'intervento di modifica ed i processi di comprensione del codice che vengono effettuati sono simili. Il processo di corrective maintenance è inoltre un processo reattivo, ovvero si istanzia un processo di corrective maintenance nel momento in cui arriva un report o un ticket che segnala un problema o un'anomalia
- **adaptive:** adatta il sistema nuove esigenze, teconologie e nuove normative. L'idea è di consentire al sistema di adattarsi al suo ambiente di elaborazione o a nuove tecnologie. Il software deve essere modificato per interfacciarsi con l'ambiente che sta cambiando o che è cambiato. Si hanno quindi driver tecnologici (nuove tecnologie) o driver di normative (cambiamento di leggi o normative). È spesso un tipo di manutenzione reattiva e non preventiva. Spesso i drive tecnologici sono driver per nuove opportunità di business, e che quindi portano alla modifica del software.
- **perfective:** miglioramento prevalentemente di requisiti non funzionali, come user experience, processing efficiency, manteinability... Si potrebbero ad esempio migliorare gli output per una maggiore leggibilità, rendere il programma più veloce. Ciò che riguarda la ristrutturazione, spesso a livello artchitetturale, prende il nome di reengineering

È stato successivamente aggiunto anche un quarto tipo di manutenzione:

- **preventive:** molti hanno usato questo termine come un **sottoinsieme di manutenzione perfettiva**, andare a migliorare la manutenibilità del software mediante ristrutturazione, ingegnerizzazione, in modo da ridurre i costi di manutenzione del software. Attualmente per **preventive maintenance si intende un'attività di manutenzione che va ad evitare che si verifichino delle failures, modificando il software in modo tale da evitare che ci siano dei problemi**. Il tipo di intervento che **rende più facile fare modifiche future**. Se non si esegue ciò, facendo interventi di evoluzione del software adattativi o perfettivi, la complessità del sistema è troppo elevata per cui si rischia di degradarne ulteriormente la qualità. Per preventive si intende attualmente un **tipo di manutenzione che cerca di prevenire l'insorgenza di failure**, ovvero intervengo sul sistema in modo da evitare che si manifestino delle failures durante l'esercizio. Per molti sistemi, a furia di andare in servizio, a causa di problemi software, si sporca la memoria producendo quindi delle failure. Si va quindi a pulire la memoria per evitare di incappare in tali failure. Altro tipo di intervento preventivo, è quello di **tipo correttivo**, in cui si va a correggere il sistema in maniera reattiva ovvero quando si verifica la failure. Essendo in emergenza si interverrà subito andando successivamente a migliorare il codice. Alternativamente si ha il **tipo preventivo**, non essendo in emergenza e potendo quindi seguire il processo di manutenzione.

Il concetto di preventive è quello di **evitare che problemi accadano**, modificando il prodotto software prima quindi che il problema si verifichi. Bisogna capire quali siano i rischi dati l'utilizzo di un software ed evitare che tali problemi si verifichino. Per fare preventive maintenance è necessario avere un **sistema di monitoring**, mediante il quale si otterranno dei log che permetteranno, mediante l'utilizzo di data mining, di **capire se ci sono dei pattern che possono portare a problemi**. Il concetto di **software rejuvenation** è una tecnica di preventive maintenance che permette di fare lo **shut down del software per fare la pulizia della memoria**, evitando, in caso, problemi dovuti alla sporcizia della stessa. Per fare ciò però si aumenta il downtime del sistema, per cui bisogna fare attenzione ai requisiti non funzionale di availability.

1.4 Software maintenance secondo Kitchenam

La suddivisione di Kitchenam, presente anche nel nuovo standard ISO/IEC IEEE 14764, divide la manutenzione in:

- **corrective:** che include come sotto categorie la **preventiva** e quella **correttiva**. Si ha quando c'è una differenza tra comportamento atteso ed effettivo
- **enhancements:** che include come sotto categoria **adaptive** e **perfective**. Ci sono a loro volta due tipi di enhancements, quello che **cambia requisiti esistenti**, quello che **aggiunge nuovi tipi di requisiti** e quello che **migliora requisiti funzionali**. L'intervento migliorativo spesso viene fatto perché sono degradate alcune caratteristiche del sistema, per cui il software non rispetta più alcuni requisiti non funzionali.

Sono stati definiti dei modelli di evoluzione e dei processi di manutenzione: **iterative models**, **staged model** e **Change mini-cycle models**.

1.5 Standard di manutenzione

L'ISO/IEC 14764 descrive il processo di manutenzione come un **processo iterativo**, composto da varie fasi. La prima è la **process implementation**. Durante questa fase si va quindi a **definire il modello di processo che si userà per la manutenzione**. Il modello scelto cambierà in base al tipo di manutenzione che si andrà ad eseguire.

Altre fasi sono **problem & modification analysis**, **modification implementation** che ingloba design, implementation, test, ovvero l'intero processo. In tale fase si decide se fare o meno l'intervento, eseguendo quindi anche l'**impact analysis**. Segue poi **maintenance review/acceptance**. Avendo prodotto una nuova baseline del software, quest'ultimo dovrà essere approvato di nuovo. Seguono poi **migration** e **retirement**.

La manutenzione è sempre stata vista come un lavoro poco scientifico, rendendosi poi conto che fosse necessario avere attività pianificate che aiutassero lo sviluppatore nella manutenzione. ISO/IEC 14764 vede la manutenzione come la totalità delle attività richieste che provvedono ad un supporto cost-effective ad un sistema software. Le attività sono eseguite durante un **pre-delivery stage** e durante il **post-delivery**. Le attività svolte durante il **pre-delivery** sono attività di pianificazione per le attività post-delivery, mentre le **post-delivery** includono modifiche al software, training ed un esercizio di helpdesk per supporto all'utente

Esempio: il modello di processo da adottare per la manutenzione correttiva, prevederà un **sistema di ticketing**. Ci sarà un primo livello di help desk che aiuta l'utente in operazioni non riesce a fare. Il secondo livello si raggiunge quando c'è un reale problema ma non a livello software, come un problema legato alla perdita di corrente. Nel caso in cui il problema

verificatosi sia ad esempio una failure software, allora bisogna intervenire sul codice, dovendo passare al terzo livello di help desk e capire il blocco in cui c'è il difetto e quindi capire quale sia la persona a cui affidare la risoluzione del problema. Quando un sistema è in servizio, i modelli organizzativi sono molto importanti.

1.6 Software configuration management

Serve a **gestire e controllare i cambiamenti dell'evoluzione software**. Durante la manutenzione, il **concepto di baseline** è fondamentale. Prodotta una nuova release del software, quest'ultima diventerà la nuova baseline, che dovrà essere **testata prima di essere effettivamente messa a disposizione**. L'**approvazione delle change baseline** è ad opera del **change control log**. Il configuration management si applica a software baselined. Esso assicura che il software rilasciato **non sia contaminato da cambiamenti non controllati o non approvati**. Un SCM (software configuration management) ha quattro elementi:

- identificazione della configurazione del software
- controllo della configurazione del software
- auditing sulla modifica effettuata per produrre una nuova baseline
- stato della configurazione del software

1.7 Reengineering

Il reengineering è l'**alterazione di un sistema software per ricostruirlo in una nuova forma e la susseguente implementazione della forma**. Si può avere il reverse engineering a diversi livelli: a **livello di codice** (estratto le rappresentazioni in termini di grafo, mediante parsing) o a **livello architetturale** (livello più alto trasformare l'architettura e produrre una nuova versione) o a **livello di requirements** (modificare i requisiti e ridurre il sistema per il reengineering). Il costo del reengineering è dovuto da diverse fasi:

$$\text{reengineering} = \text{costo del reverse engineering} + \text{costo del delta} + \text{costo del forward engineering}.$$

Per delta si intende il costo di trasformazione del modello ottenuto dal reverse engineering. Il reengineering prevede quindi diverse fasi. Rispetto al **refactoring**, che **altera il sistema ma non ne modifica le funzionalità**, il **reengineering può modificare anche le funzionalità se si spinge fino al livello dei requisiti**, anche se il reverse engineering si applica soprattutto a **livello architetturale**.

1.8 Legacy Systems

Sistemi vecchi importanti per l'azienda che li detengono. Questi sistemi si chiamano **legacy** poichè coloro che ci lavorano attualmente non sono gli stessi che li hanno sviluppati, per cui sono **sistemi ereditati**. Ciò che non verrà fatto è un redevelopment, poichè altamente rischioso.

1.8.1 Strategie dei legacy systems

Alcune strategie sono:

- **Freeze:** Nel frattempo che si cerca un sistema nuovo, **il sistema attualmente usato non lo si evolve più**. Vado a sostituire man mano, con dei componenti, le funzionalità nuove che dovrebbero essere necessarie
- **Outsource:** un'organizzazione potrebbe decidere che **supportare un software non è un obiettivo primario**
- **Wrapping:** possibile farlo anche senza freezing, per wrapping si intende **incapsulare il sistema per renderlo utilizzabile in un nuovo ambiente operativo**. Lo si incapsula per consentire allo strato di middleware (wrapper) di gestire la comunicazione tra la parte nuova del sistema con la parte legacy
- **Migrate:** essa comporta, spesso, dei sistemi di wrapping prima di essere rimpiazzato. In questo caso **il discarding non è mai in un solo passo**, ma tutti i passaggi sono incrementali. Nel frattempo che **il sistema viene sostituito un pezzo alla volta**, passa del tempo. Seppur lungo come processo è però meno rischioso di risviluppare il sistema da 0

1.9 Impact analysis

Si stimano le parti del software che possono essere influenzate da una modifica. È quindi necessario individuare le parti che devono essere modificate per implementare la modifica e poi le componenti che possono subire effetti collaterali. Sono quindi utili i link di tracciabilità tra componenti a diversi livelli di astrazione e sono inoltre importanti le analisi di dipendenze all'interno del codice. Viene fatta una **stima dei costi per effettuare questa decisione**. Stima le parti necessarie per modificare l'algoritmo e stima le parti del software che possono essere influenzate da una modifica (effetti collaterali).

1.10 Refactoring

Termine usato per il **reengineering fino a livello architetturale** o livello di ristrutturazione nel contesto dell'object orientation. Il refactoring a differenza del reengineering **deve però mantenere le funzionalità del sistema**, non arrivando ai requisiti e rimanendo alle modifiche sul codice. Nel refactoring si usano però **tecniche di reverse engineering, tecniche di modifica delle rappresentazioni**, ma sono sempre tecniche che si fermano al livello architetturale. Esse servono fondamentalmente a migliorare la struttura interna e quindi a migliorare la manutenibilità del sistema.

1.11 Program comprehension

La program comprehension va a **mappare concetti sul codice**. Il linguaggio usato nella richiesta di modifica è quello dei requisiti, linguaggio quindi molto vicino all'utente. È quindi **necessario localizzare tali concetti all'interno del codice, capire dove sono implementati, per andare a modificarli**. Esistono quindi diverse tecniche per lavorare sulle componenti di questi concetti: top down, bottom up... Mediante tecnica top down, scendendo si vanno a creare delle ipotesi. Si costruiscono quindi **modelli mentali del sistema software a diversi livelli di astrazione**.

1.12 Software reuse

Il concetto di riuso del software si basa sul fatto che **il riuso può migliorare la produttività del software**. Nato come spinta per avere una maggiore produttività per la produzione di massa del software. **Non si riusa solo il codice ma anche altri aspetti del software**. Anche fra i requisiti del software c'è la riusabilità. Un componente è riusabile quando è **comprendibile, affidabile, adattabile e portabile**.

CAPITOLO 2

Processo di manutenzione

2.1 Introduzione: processo di manutenzione

Sul processo di manutenzione va ad impattare il prodotto che viene manutenuto, il tipo di manutenzione che viene fatta, le persone ed i processi della maintenance organization.

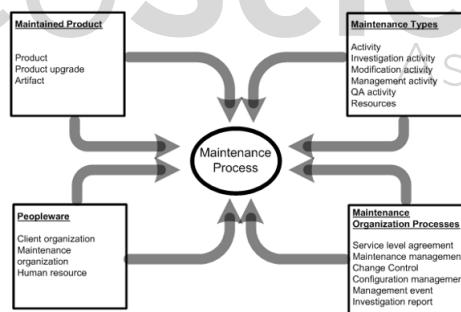


Figura 2.1: Processo di manutenzione

2.2 Prodotto

Il prodotto mantenuto si divide in:

- **prodotto**: visto come una **collezione di diversi artefatti**.
- **Product upgrade**: in che modo è mantenuto il software all'interno di una repository, **come sono mantenute le baseline e come sono aggiornate**. La parte di configuration management è importante nell'ambito del prodotto software.

- **Artifacts:** Artefatti di cui è composto il prodotto come codice, documenti... è importante la dimensione degli artefatti, qualità, composizione..

2.3 Tipi di manutenzione

Ci sono diversi tipi di maintenance activities:

- **Investigation:** si va a **investigare il software** per capire **che impatto avrebbe una certa modifica**
- **Modification:** fase di **modifica vera e propria**
- **Management:** questa fase riguarda **configuration control...** Quindi non supporto in termini di versioning, ma proprio **come è gestito il processo di controllo dei cambiamenti.**
- **Quality assurance:** come controllo che la **qualità di quello che vado a modificare sia conforme.**

Si ragiona in termini di processi di manutenzione e processi di supporto alla manutenzione.

2.4 Processi della maintenance organization

Ci sono due livelli dei processi di manutenzione:

- **individual-level:** nel momento in cui si decide di fare una modifica essa viene assegnata a qualcuno che seguirà un processo per effettuare la modifica
- **organization-level:** gestiscono a livello più alto, **dalla change request alla chiusura della stessa.** Ad un certo punto di questo livello organizzativo partirà un processo individuale di modifica. Gli elementi più importanti di questo livello sono:
 - **Event management:** non si parla di eventi che accadono ad un sistema software nel corso della sua esecuzione, ma di una **coda di ticket che segnalano un problema,** o una **coda di change request da prioritizzare.** Nella **manutenzione correttiva** i ticket che arrivano vanno gestiti con la **teoria delle code**, per cui sarà da tenere in conto il rate di soluzione, ovvero quanto tempo ci impieghiamo per risolvere il problema, e il rate di arrivo delle richieste. Nel caso in cui si parli di **manutenzione evolutiva**, sarà necessario pianificare **cosa mettere nella prossima release** e quindi **cosa andare a sviluppare in un certo lasso di tempo.**

- **Configuration management:** modo in cui gestisco le configurazioni
- **change control system:** modo in cui vado a **valutare le manteinace request e decidere se andarle ad implementare o meno.**

Le change request possono provenire da clienti, utenti e maintainers. Dagli **utenti** arrivano prevalentemente **richieste di manutenzione correttiva** o **suggerimenti di miglioramento dell'usabilità del prodotto**. Dai **clienti** arrivano invece **richieste che servono a supportare nuovi processi**, come aggiunta di funzionalità per nuove possibilità di business o per **supportare delle parti che il sistema non supporta**. Dai **maintainers** potrebbero provenire **richieste di miglioramento della manutenibilità**, poiché loro si rendono conto dei costi della manutenzione.

Si fa un'investigation iniziale e si mette in piedi un **management process per approvare le attività di change** (ad opera di un change control board). Una volta approvata è **necessario definire un SLA**, accordo con gli utenti, che definisce in quanto tempo deve essere ripristinato l'esercizio del sistema, ad esempio su quanto possa essere in downtime il sistema... SLA è un qualcosa comune anche nelle applicazioni service based, in cui non si ha un software ma si ha qualcuno che distribuisce servizi software, per cui è necessario definire una serie di clausole.

2.4.1 Livelli di help desk

La manteinace organization process usa 3 livelli di servizio:

- **Livello 1:** Raccoglie i problemi ed identifica le persone che possono assistere alla persona che riporta il problema. Si **assiste la persona nel formulare il problema, smistando alla persona che può risolvere il problema**. In questo caso non parte ancora il ticket
- **Livello 2:** Se l'affiancamento del primo livello non è bastato, si arriva al secondo livello, che **analizza il problema raccomandando quick fixes e temporary workaround**. Se l'utente non riesce a risolvere il problema, nel caso in cui il problema sia non causato dal codice ma bensì causato dallo stato del database, problema sul server... si prova ad intervenire.
- **Livello 3:** Si passa a questo livello **quando il problema è un problema software**. Il livello **include i programmatore che possono effettuare modifiche sul prodotto software**. Parte ora il processo di manutenzione.

Il ticket può essere risolto o perché lo **stato del sistema è inconsistente**, per cui si può **intervenire senza fare modifiche al codice** (livello 2), o **bisogna far intervenire lo sviluppatore** (livello 3)

2.5 Peopleware

Ai fini della manutenzione **non si possono ignorare gli elementi umani**, in quanto software production e maintenance sono attività intensive da punto di vista umano. Ci sono tre **concentri people-centric**:

- **Maintenance organization:** organizzazione che **mantiene il prodotto**
- **client organization:** client organization che **usa il sistema** che viene manutenuto
- **human organization:** **risorse umane sia della client che della maintenance organization**

C'è bisogno di un allineamento dal punto di vista di maturità dei processi tra le due organizzazioni. Molto spesso la **client organization ha delle persone che sono esperte di sistemi**. Hanno bisogno di un **supporto esterno per un lavoro di manutenzione** ma sono comunque **in grado di capire cosa stanno facendo**.

2.6 User & Customer issues

Alcuni aspetti importanti che potrebbero creare problemi sono:

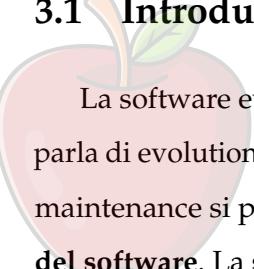
- **size:** La dimensione della customer base è importante. Quanti sono i clienti, il numero di licenze
- **variabilità:** nel caso in cui ci siano persone molto eterogenee nella customer base ci sarà **molta più difficoltà nell'interagire**
- **Obiettivi comuni:** afferiscono agli **obiettivi che possono avere utenti e customer**

Se i customer fanno **intervenire gli utenti nella SLA** si avrà un **servizio di manutenzione adeguato anche alle esigenze degli utenti**, altrimenti no

CAPITOLO 3

Evoluzione del software

3.1 Introduzione: Evoluzione del software



La software evolution è usato per definire la **crescita dinamica del software**. Quando si parla di evolution si fa riferimento a ciò che accade al software, come esso evolve, mentre con maintenance si parla di un processo. Gli interventi di maintenance causano l'**evoluzione del software**. La software evolution può essere visto come un qualcosa che prende un input e produce un output. L'**input** è il sistema da cui si parte mentre l'**output** è il sistema dopo l'**evoluzione**. La **maintenance** è usata per il **supporto post delivery**, ovvero il supporto che consente di mantenere in esercizio il sistema, mentre l'**evoluzione** riguarda interventi migliorativi guidati da cambiamenti nei requisiti. L'**evoluzione** quindi riguarda lo stato del software, mentre la **maintenance** riguarda le attività ed i processi che vengono eseguiti durante l'**evoluzione**.

3.2 Approcci alla software evolution

Software evolution è stato studiato in due modi:

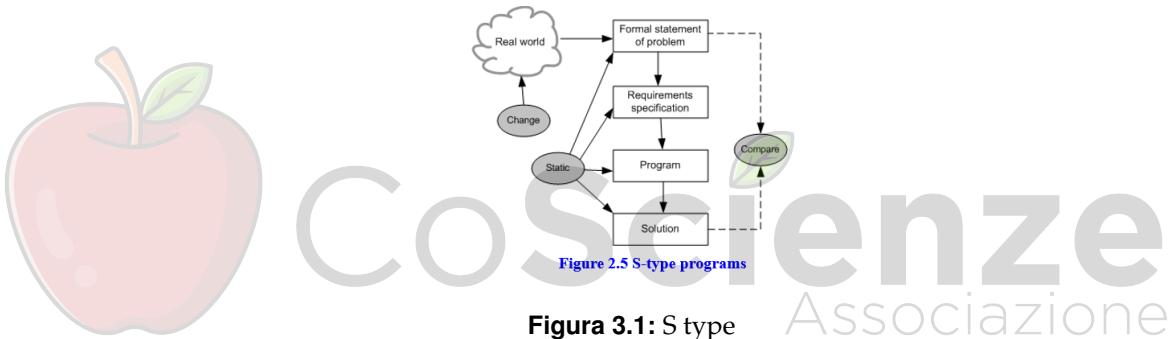
- **studi descrittivi (explanatory)**: approcci che cercano di spiegare le cause della software evolution, i processi usati e gli effetti sul software. Un esempio è la ricerca di code smell (problemi di qualità interni nel codice). Volendo fare un studio sul come e perché evolvono i code smell, si andrà a fare uno studio di tipo explanatory.

- **process improvement:** questo approccio tenta di gestire gli effetti del software evolution sviluppando metodi e strumenti migliori. Si cerca di usare studi empirici per capire se possono essere sviluppati dei tool che consentono di fare un determinato lavoro più agevolmente, per migliorare la produttività o facilitare operazioni da parte di uno sviluppatore. L'evoluzione viene studiata da un punto di vista ingegneristico, proponendo soluzioni ingegneristiche come metodi e tool.

3.3 SPE Taxonomy

Sono stati definiti 3 tipi di sistemi:

- **specified:** Tutte le proprietà funzionali e non funzionali sono formalmente completamente specifiche. La correttezza del programma, rispetto la sua specifica globale, è l'unico criterio di accettabilità della soluzione agli stakeholder.



Esempio: Specificato un software matematico, pur cambiando il mondo reale, il modulo non cambierà poiché non cambia ciò che il software deve fare.

- **problem:** basato su un'astrazione pratica del problema piuttosto che basarsi su una specifica completa del problema. Cambiando il mondo reale e quindi migliorando le tecnologie, potrebbero esserci tecnologie che permettono l'utilizzo di operazioni più accurate. **Cambia l'implementazione ma non cambia cosa il software fa.** Non è infatti una soluzione nuova ma è una modifica della soluzione vecchia
- **evolving:** programma embedded nel mondo in cui si trova, per cui cambia insieme al mondo reale. Questi sistemi vengono visti come sistemi di feedback. Lo sviluppo può portare a dei cambiamenti sul dominio. Si potrebbe quindi cambiare qualcosa il che potrebbe portare dei cambiamenti al problema, e viceversa. L'utente potrebbe suggerire delle modifiche poiché usa il programma, ma anche l'apportamento di una nuova feature potrebbe portare all'esigenza di una nuova funzionalità. Si ha quindi

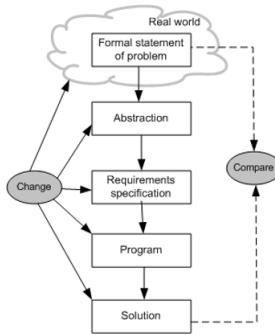


Figure 2.6 P-type programs

Figura 3.2: P type

un loop continuo, cosa che invece non succedeva prima. I cambiamenti quindi possono provenire non solo dall'esterno ma anche dall'interno

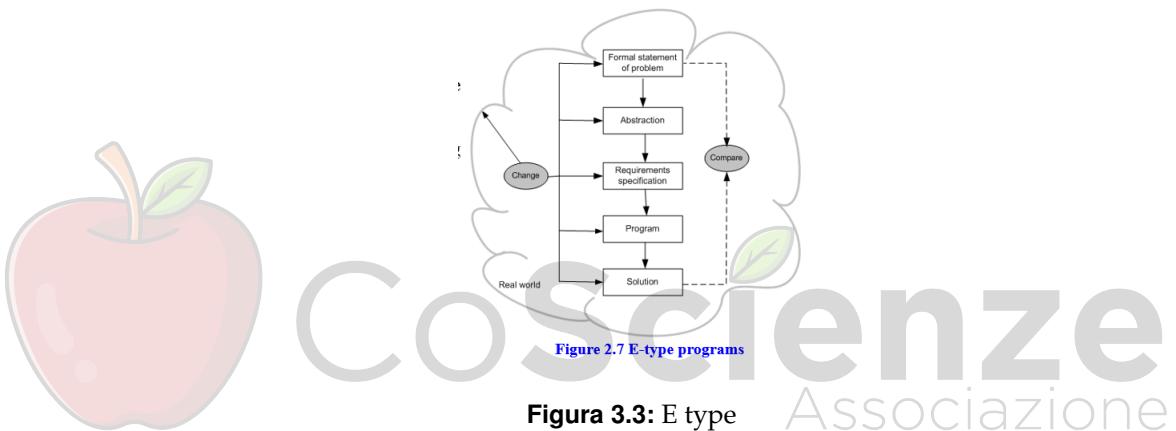


Figure 2.7 E-type programs

Figura 3.3: E type

Si hanno quindi differenze tra **software sviluppati per avere un insieme fisso di requisiti**, e **software sviluppati per risolvere problemi reali che cambiano nel tempo**.

3.4 Leggi della software evolution

Fenomeni osservati mediante studi empirici. Questi fenomeni sono **intrinsechi nel software e non sono influenzati da decisioni di manager o sviluppatori**. I sistemi sono stati visti come entità la cui evoluzione è indipendente da ciò che le persone che fanno manutenzione vogliono, **dipendendo maggiormente dall'ambiente in cui operano**. Queste leggi afferiscono principalmente a **sistemi funzionanti su mainframe, sistemi monolitici** prodotti da un team all'interno di un'organizzazione che risolvono un problema reale e hanno utenti. Le leggi sono otto:

- **continuous change:** i programmi devono essere continuamente cambiati altrimenti diverranno meno utili, all'interno dell'ambiente. Se cambia la realtà ma non cambia il sistema, ad un certo punto si avrà un disallineamento tra realtà e sistema, diventando inutili. Più i sistemi diventano utili più aumenta la probabilità per i sistemi di venire dismessi ed essere sostituiti da altri prodotti
- **increasing complexity:** man mano che un programma E-type evolve, la complessità cresce fin quando non si fanno lavori mirati per diminuire o ridurre la complessità. La complessità cresce in quanto si vanno ad aggiungere funzionalità, che potrebbero degradare la qualità del prodotto. Tutti i cambiamenti sono aggiuntivi poiché non dipendono dalle esigenze dall'ambiente che cambia, ma sono esigenze legate alla riduzione dei costi di manutenzione. Per arrestare il declino della qualità software devo quindi intervenire con operazioni mirate
- **self regulation:** l'evoluzione del software è self regulated, con la distribuzione nel tempo di misure di processi e prodotti vicina alla normalità. è come se in ogni periodi di manutenzione si impiegasse lo stesso tempo di lavoro. Ciò è dato dal fatto che le aziende sono molto stabili nel numero di lavoratori e nel numero di ore, per cui il lavoro di manutenzione risulterà avere un tempo stabile. Misure di prodotto e di processo sono distribuite normalmente come risultato di un grosso numero di decisioni manageriali.
- **conservation of organizational stability:** la quantità delle persone che lavora ad un progetto software è più o meno identica. Le risorse utilizzate nello staffing hanno un piccolo effetto sul long term evolution. Per questa e la terza legge si ha un'indipendenza dalle decisioni manageriali.
- **conservation of familiarity:** questa legge suggerisce il fatto che nel passare da una release all'altra non si può cambiare molto in quanto gli sviluppatori hanno bisogno di mantenere una familiarità con il sistema. Non cambiando molto il sistema ed avendo una stabilità organizzativa, si conserva una familiarità con il sistema. Non si aggiungono molte features per mantenere familiarità con il sistema. Implica che ingegneri e sviluppatori devono avere lo stesso livello di understanding di una nuova release anche se più funzionalità sono aggiunte.
- **continuing growth:** il cambiamento porta ad una continua crescita del sistema, aumentando funzionalità, size... Questa e la prima legge rappresentano concetti simili

li, in quanto la prima dice che il sistema deve cambiare mentre questa dice che il cambiamento è un **cambiamento che porta ad una continua crescita del sistema**.

- **declining quality: degrado dell'affidabilità o delle performances.** Si è usato il concetto di entropia per questa complessità crescente, associato anche con **code decay o software aging**. Legata in parte alla seconda. Mentre in quel caso si parla di declino legato alla complessità crescente, in questa si affronta maggiormente l'**aspetto del declino della qualità anche dal punto di vista di performances ed efficienza del sistema**. È stato usato il concetto di **entropia per esprimere la complessità crescente**. Il declino della qualità del software è anche legato al product aging o al code decay
- **feedback system:** l'evoluzione dei processi E-type costituisce **multi-agent, multi-level e multi-loop feedback**. Si avrà quindi un **processo iterativo** (multi-loop). Multi-level fa riferimento al fatto che **occorre in più di un aspetto del software della sua documentazione** e multi-agent indica che il **software è fatto da diversi agenti che cooperano** ai fini di ottenere task individuali o collettivi ed il feedback determina il modo in cui questi agenti comunicheranno. Il feedback che mi da una modifica ad un modulo fornisce uno spunto per modificarne anche un altro modulo

In queste leggi non rientrano i P-type in quanto **tali programmi non cambiano sulla base della realtà che cambia**. Posso decidere di aggiornare un P-type in quanto posso avere una soluzione più performante, ma non cambia con la realtà in cui si trova. Le **prime due leggi sono vere in assoluto**, la **terza, la quarta e quinta** funzionano solo all'interno dei sistemi analizzati, ovvero **mainframes**, ambienti in cui le cose non possono cambiare molto. Esse però sono più leggi che afferiscono alle organizzazioni che coinvolgono il processo di manutenzione che il software stesso. Da tenere in considerazione che il **software che gira sul mainframe ha delle limitazioni fisiche**, per cui un programma non può crescere a dismisura. La **prima e la sesta legge riflettono fenomeni legati tra loro**. Sono poi stati studiati altri sistemi per verificare tali leggi. Si è osservato che la curva di crescita può essere lineare o meno che lineare, si è poi verificato che **1,2,3,5,6,8 erano leggi valide** mentre la "declining quality" era **basata solo su analisi teorica** mentre la "conservation of organizational stability" non è né supportata né classificata.

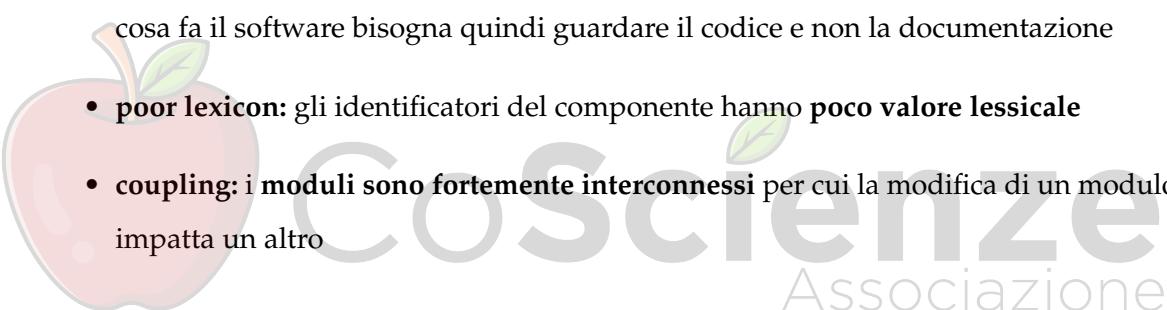
3.5 Aging

Esistono vari tipi di aging:

- **aging dal punto di vista di esecuzione:** si ha un **degrado delle performances** o un **aumento delle failures**
- **software product aging:** si **manifesta all'interno del codice**, analizzandolo da un **punto di vista statico**

I vari sintomi di aging sono:

- **pollution:** ci sono dei **moduli che stanno nel software ma che non vengono utilizzati**, magari perché non vengono aggiornati. Il problema è che facendo fare manutenzione da terze parti, essendo che i costi di manutenzione vengono stimati sulla base dell'intero software, vuol dire che **ho del codice in manutenzione che pago ma che non uso** per cui non sarà mai modificato
- **embedded knowledge:** ho la documentazione ma ci sono alcune **cose che non ricavo da nessuna parte per cui non c'è documentazione**, ovvero c'è solo il codice. Per sapere cosa fa il software bisogna quindi guardare il codice e non la documentazione
- **poor lexicon:** gli identificatori del componente hanno **poco valore lessicale**
- **coupling:** i **moduli sono fortemente interconnessi** per cui la modifica di un modulo ne impatta un altro



3.6 Code decay

Si capisce se il **codice è decaduto dal costo dei cambiamenti**. Più tempo o costo mi impiega maggiore sarà il decadimento del codice. **Con ogni modifica che verrà fatta si andrà quindi a peggiorare la qualità.**

3.7 COTS based systems

I COTS sono componenti che compro e di cui non si ha il codice. Sono utilizzati all'interno di sistemi. Si utilizzano i COTS per **riutilizzare software esistenti e ridurre quindi i costi di sviluppo**. Mi aspetto che la qualità sia alta ma non posso modificarli, in quanto **utilizzati in modalità black box**. Nel caso in cui la componente non fa ciò che si vuole, possiamo utilizzare alcune tecniche come l'adapter. Esso crea un'interfaccia che rispecchia le funzionalità che io voglio. L'adapter eredita l'interfaccia della componente che vorrei sviluppare. Si specializza

l’interfaccia con una classe adapter che delega alla componente COTS l’implementazione della funzionalità.

Nel caso in cui il COTS faccia più di quanto si vuole, con il **wrapper** posso utilizzare del componente COTS, **solo ciò che mi serve**. Nel caso in cui i linguaggi del COTS e del sistema siano differenti il **wrapper permette di mascherare il COTS in modo da utilizzarlo in un determinato modo**. Con la **glue** è possibile **mettere insieme le funzionalità di più componenti COTS**. Si potrebbe fare **tailoring** nel caso in cui il **componente COTS non è sufficiente a fare tutto ciò che serve, dovendo aggiungere funzionalità**. Si potrebbero prendere delle funzionalità dal componente COTS, prende i risultati e li va elaborare ulteriormente per fornirli nel modo in cui servono.

3.7.1 Maintenance dei CBS

La manutenzione dei COTS Based Systems (CBS) è difficile in quanto le **funzionalità sono freezate**, ci possono essere **incompatibilità di upgrade**, dovendo stare quindi attenti alla versione da utilizzare. Potrebbe essere necessario infatti fare manutenzione del software a causa di **third part libraries**. Si possono avere problemi legati alla **mancanza di codice**, problemi legati a **componenti non affidabili o middleware difettosi**.

CAPITOLO 4

Ciclo di vita del software

4.1 Introduzione: ciclo di vita del software

Fondamentalmente in un **SDLC** (software development life cycle) ci sono due fasi: una di **sviluppo** con il proprio ciclo di vita, ed una di **manutenzione**, ciclica a sua volta poichè effettuabili diversi interventi di manutenzione.

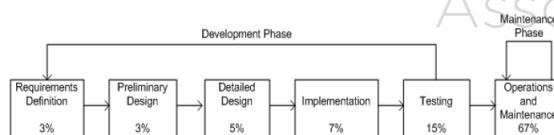


Figura 4.1: ciclo di vita del software

Il costo della manutenzione è circa **2/3 del costo totale**, questo poichè **il software vive molto più a lungo in fase di manutenzione** (operation e maintenance) che non nella fase di sviluppo. La fase di sviluppo sarà infatti relativamente breve.

La manutenzione è diversa dallo sviluppo a causa di:

- **vincoli sul sistema esistente:** quello che andremo a manutenere è un software in esercizio. Come tale, ci saranno dei **vincoli sul modo in cui possiamo fare manutenzione**. **Non si può rivoluzionare il sistema ogni volta che si fanno operazioni di maintenance.** Il motivo per cui si ha il degrado della seconda legge di Lehman è proprio perché **non posso mettermi in un ottica di sviluppo in cui si riprogetta interamente**

un'architettura e la si rilascia. Ciò potrebbe avere tempi troppo lunghi rispetto alle esigenze di servizio

- **time frame più corto:** nel caso di **manutenzione correttiva** si dovrebbero impiegare **delle ore**, arrivando poi ad alcuni mesi a differenza del tempo impiegato in fase di sviluppo per progetti grandi, tempo che potrebbe arrivare anche a diversi anni
- **disponibilità dei test data:** sviluppando un sistema da 0 devo sviluppare anche i test mentre nel caso di manutenzione ci sono già i test, per cui è necessario adottare delle **strategie per effettuare regression testing esaustivo**, piuttosto che rieseguire tutti i test.

Visto che la durata del ciclo di vita è prevalentemente **manutenzione e operation**, è evidente che la manutenzione è un'attività, ma è ansi necessario pensare a modelli di ciclo di vita del software durante l'esercizio della manutenzione che siano a sé stanti. La **manutenzione dovrebbe avere il suo ciclo di vita**, che insieme al **ciclo di vita dello sviluppo** forma il **ciclo di vita del software**.



4.2 Cicli di vita per la manutenzione

Essendo il ciclo di vita prevalentemente manutenzione, si è iniziato a pensare dei **cicli di vita per la manutenzione**, differenti dal normale ciclo di vita dello sviluppo software:

- **quick fix model:** I cambiamenti necessari sono **fatti velocemente**, si **modifica prima il codice sorgente e poi si modifica il resto**. Questa tecnica ricorda la **manutenzione correttiva di emergenza**. Le modifiche vengono fatte senza **un'impact analysis accurata**, dovendo successivamente intervenire in caso di difetti. Generalmente questo tipo di approccio non va bene per operazioni di enhancement

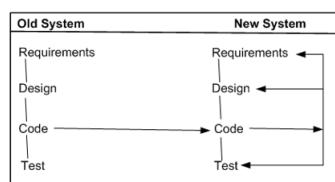


Figura 4.2: Fix model

- **iterative enhancement model:** i cambiamenti sono fatti **prima sui documenti di alto livello**. Si fa l'**impact analysis** partendo dalle **change requests** che sono espresse in

linguaggio naturale. Le **change requests** definiscono cosa il software dovrebbe fare, quali funzionalità dovrebbe aggiungere. Si parte quindi **dai requisiti fino a scendere e progettare l'intervento**, tenendo conto dell'architettura esistente del sistema. Per cui, si ha un sistema, arriva una richiesta che viene analizzata, si passa poi al code, test, design...

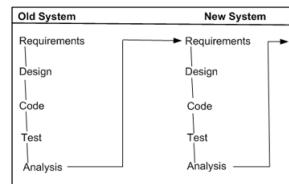


Figura 4.3: Iterative model

- **full reuse model:** si è pensato allo sviluppo software come un qualcosa che andasse ad **assemblare componenti piuttosto che svilupparle**. Man mano che si sviluppano cose nuove, esse verranno messe in un repository in modo tale che possano essere poi riusate. Con il **primo sistema che sviluppo** quindi popolo la repository, con il **secondo sistema, identifico le componenti del vecchio sistema che possono essere riusate**. Verranno al più modificate per comprendere i nuovi requisiti. In questo caso il **riuso non afferisce solo al codice ma anche alle specifiche, design...** Le componenti identificate possono essere poi modificate per meglio adattarsi alle esigenze. Questa tecnica però non ha preso molto piede in quanto c'è molta più iterazione tra i diversi ruoli all'interno di un team di sviluppo, per cui una **tecnica simile è impensabile**. Pur non essendo pensabile ad una strategia simile, i **concetti di riuso sono validi**, e mettere insieme un programma di riuso, è un qualcosa di utilizzato. Non tutto però è riusabile. Pensando ad applicazioni in uno specifico dominio applicativo, è difficile pensare a produrre un qualcosa di riutilizzabile. Per poter utilizzare delle componenti in uno specifico dominio applicativo, sarebbe infatti necessario dover sviluppare più applicazioni, o **avere a disposizione più applicazioni**

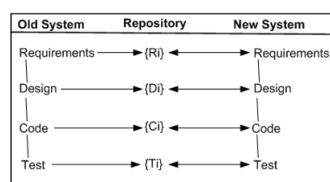


Figura 4.4: Full reuse model

4.3 Sviluppo iterativo e sviluppo incrementale

Si ha differenza tra sviluppo iterativo e sviluppo incrementale:

- **Iteration:** Si ha un **processo ciclico**, per cui si **ripetono delle operazioni in maniera iterativa**.
- **Iterative:** basato su strategie di scheduling in cui vado a **rivedere e migliorare parti sotto sviluppo**. Ciò che si vuole sviluppare è stato già sviluppato. **Si va a migliorare un prodotto** per migliorare il sistema o perchè sono richieste nuove esigenze e nuove funzionalità. **Le funzionalità aggiunte però non sono previste fin dall'inizio**. Nell'iterativo devo necessariamente rivedere in qualche modo ciò che già sviluppato, per migliorarlo. C'è una **fase di restructuring del codice, non presente però nel modello incrementale**. Se il modello fosse solo iterativo, si avrebbe un sistema ed ogni volta che si vuole fare un qualcosa **si va a migliorare un aspetto del software aggiungendo, eventualmente, nuove funzionalità**. In un processo iterativo s'è sempre il concetto di **riorganizzazione del codice, che non c'è in un modello di processo incrementale puro**.
- **incremental:** quello che dovevo sviluppare lo **sviluppo a pezzi rilasciando poco per volta**. L'architettura tiene già in conto di cosa si vuole aggiungere. **Devo solo aggiungere quello che era stato già prestabilito intregnandolo con quanto è stato già sviluppato**. Tutte le funzionalità sono quindi definite dall'inizio. In un modello puramente incrementale tutte le features del sistema sono già definite dall'inizio, sviluppando solo il sistema in più fasi. Il modello incrementale potrebbe essere visto come una **variante del modello a cascata in cui invece di sviluppare tutto insieme, lo sviluppo a strati**. Fino alla fase di progettazione architettonica, l'applicazione è progettata interamente, non essendoci però nuove funzionalità. I **modelli incrementali quindi nascono come puri ma diventano poi anche iterativi nel caso in cui emergano nuove esigenze**.

Non esiste un modello puramente iterativo o puramente incrementale. Si hanno dei requisiti e partendo da questi requisiti viene fatta una progettazione e una volta sviluppato il sistema bisogna magari aggiungere un qualcosa e contemporaneamente migliorare parte del sistema. è possibile che sia necessario aggiungere nuove parti, ma **nel frattempo potrebbero emergere nuovi obiettivi, per cui il processo è sì incrementale ma difficilmente non è anche iterativo**. I **modelli incrementali**, poichè rilasciano il sistema a pezzi, **riducono i rischi**

4.4 Gli staged models di CSS

Rajlich e Bennett hanno presentato un **modello di software evolution chiamato staged model of maintenance and evolution**. Il modello è diviso in 4 fasi:

- **initial development:** la **prima versione delivered del sistema**. La conoscenza del sistema è fresca e i requisiti cambiano spesso, fin quando non si stabilizzano e fin quando non emerge un'architettura con cui posso organizzare un'applicazione, implementarla e rilasciarla
- **evolution:** **cambiamenti semplici sono facilmente eseguibili**, ma è possibile effettuare anche **major changes**, come enhancements dell'applicazione. I **costi ed i rischi sono maggiori** alla fase precedente in cui **non si aveva ancora il codice o non si aveva il sistema in esercizio**. Essendo in questa fase il sistema in esercizio, i rischi e costi sono maggiori. La conoscenza del sistema è ancora buona, ma è possibile però che una parte degli sviluppatori della fase iniziale non ci siano più. Per la maggiorparte dei sistemi, il grosso del ciclo di vita è speso in questa fase.
- **servicing:** Il **sistema non è più una key asset per i developers** che si concentrano solo sui task di manutenzione, per cui **il sistema non viene più evoluto ma solo manutenuto**, per continuare a mantenerlo in esercizio. La **conoscenza del sistema è diminuita** e i **cambiamenti e costi del sistema sono difficili da predire**. La fase di servicing porta ad una phase out
- **phase out:** vado a **sostituire il sistema con un altro sistema**. è possibile utilizzare un'**exit strategy** in cui si può sostituire il sistema o iniziare un processo di migrazione, in cui **si wrappa il sistema vecchio e si vanno a cambiare le componenti** man mano. Quando la fase di sostituzione è terminata allora il sistema vecchio non è più utilizzato. La fase di phase out può essere effettuata in vari modi, come **il replacement che coinvolgerà data migration** e anche il replacement non è un qualcosa che viene fatto immediatamente. Per migrare i dati dal sistema vecchio al sistema nuovo, è necessario che **entrambi i sistemi siano in esercizio**. Bisogna usare gateway, wrapper, in modo tale da migrare incrementalmente i dati da una parte all'altra

Per effettuare una **migration si impiegherà molto tempo**, per cui non è un passaggio immediato. In alcuni sistemi si è notato come alcune versioni vengano messe in evolution ed altre versioni in servicing, per cui l'**evoluzione di un sistema software non è lineare**

4.5 Staged model per FLOSS

Lo staged model per FLOSS (gli **open source**) è stato notato che **dalla fase di servicing si tornava alla fase di evolution**. Ciò è dovuto al modo in cui è organizzato il sistema. C'erano nuovi sviluppatori che davano nuovo impulso al sistema, in modo tale da portare nuovi contributor aggiungendo quindi nuove funzionalità ai sistemi che si trovavano in una fase di servicing

4.6 Change Mini-Cycle Model

Ciclo in cui **si vede il processo di manutenzione**. Viene sottomessa una change request, per cui viene **analizzato e formulato un plan change**. Si analizza la propagazione di cambiamenti, si verifica, convalida e si va a verificare che l'**impact analysis** sia stata correttamente eseguita. Esso è un **modello di processo di manutenzione**

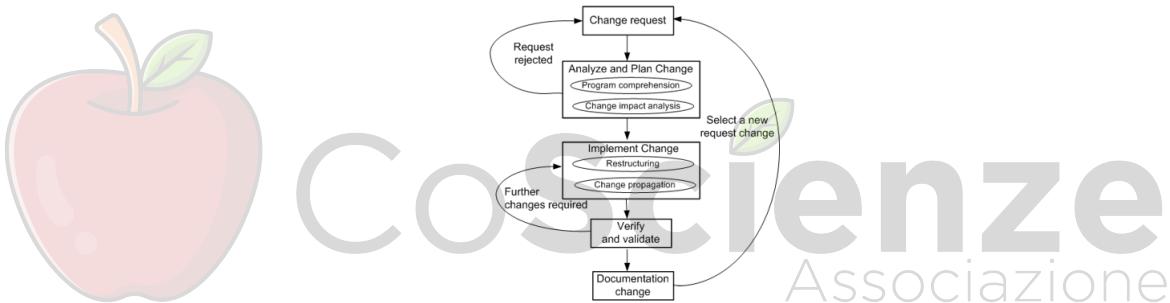


Figura 4.5: Change Mini-Cycle Model

Negli standard **non viene detto in che ordine vengono svolte le attività**, non avendo modelli di processo di manutenzione, come quello del change mini cycle model, perchè non viene riportato l'ordine delle operazioni da eseguire, ma si hanno dei modelli di ciclo dell'evoluzione del software.

Negli standard infatti, non viene dato un ordine preciso in cui le attività devono essere svolte, **ma si va solo ad indicare quali sono le attività da svolgere**. Negli standard infatti **non si specifica il tipo di modello da adottare per la manutenzione**, se **iterativo**, se a **cascata**... ma solo quali sono le attività che devono essere fatte. Verranno infatti fornite delle **linee guida ovvie**, come ad esempio il fatto che alla fine della modifica debba essere fatto review acceptance. Il **seven phase model** è un modello di ciclo di evoluzione del software. Esso esprime le fasi in cui si troverà il sistema. Le fasi sono: problem identification, analysis, design, implementaion, system test, acceptance test e delivery.

4.7 Gli standard della software maintenance

Software maintenance è definito come la **modifica di un prodotto software o componente, dopo il rilascio allo scopo di correggere fault o bug, migliorare attributi del sistema** come le performances e adattarsi a nuovi ambienti. Il sistema è **supportabile** se è **facile dare supporto al sistema durante l'operation** (durante l'esercizio).

Nello standard IEEE/EIA 1219 ci sono 7 fasi mentre lo standard ISO/IEC 14764, standard più generale, **descrive il processo in modo iterativo**, specificando che deve essere prima fatta la fase di **problem and modification analysis** e poi **modification and implementation**, ovvero operazione intuitiva. Sarà infatti ovvio dover **eseguire le operazioni in un determinato modo** in quanto output di una fase sono input di un'altra, **ma le attività da effettuare nelle singole fasi saranno scelte da coloro che dovranno effettuare la manteinance e non definite dallo standard**. Le attività non possono essere definite da uno standard poichè **dipendenti dal tipo di manteinance che bisogna effettuare**. Il tipo di processo da implementare sarà però definito dalla process implementation.

Per ogni tipo di intervento che farò necessiterò della progettazione del processo. Per questo motivo, **migration & retirement** sono due processi a parte. **Ogni attività è suddivisa in tasks**. Un sistema è **supportabile** se posso facilmente supportarlo durante l'esercizio. se il sistema è **insupportabile** vuol dire che è **complesso**, magari costoso, fornire supporto al cliente che lo utilizza. Questo standard è ormai obsoleto. **Lo standard ISO/IEC 14764 definisce invece la software manteinance come l'insieme di attività richieste per fornire supporto cost effective al sistema**. Per questo standard viene considerato un tipo particolare di manutenzione correttiva, la **manutenzione di emergenza**. Essa rappresenta una particolare tipo di operazione non schedulata per mantenere in vita il sistema, parlando in questo caso di **pending corrective manteinance**. La manutenzione nello standard 14764 è molto più generale. Supponendo che un sistema abbia un difetto, se il difetto è scoperto in tempo, prima ovvero che si generi una failure durante l'esercizio, allora **si può intervenire in modo preventivo, altrimenti bisogna intervenire in modo reattivo** con manutenzione correttiva.

4.7.1 IEEE/EIA 1219

Lo standard è suddiviso in 7 fasi. Esso è **sequenziale perché il prodotto dell'analisi va in input al design, ma non bisogna necessariamente seguire le fasi**. Indipendentemente da come si è organizzati però, è ovvio che una fase di analisi debba essere fatta prima di una fase di implementazione, **ma il modo in cui ciò è fatto su tutto il sistema non è detto**

che sia a cascata, ovvero che si faccia una fase di analisi, una fase di design, una fase di implementazione... Il modo in cui infatti le operazioni vengono eseguite dipende dal modello di processo, che in questo caso non è definito. Si ha un livello di dipendenza solo perché specifica che il design dipende da ciò che viene fatto durante la fase di analisi. Lo standard IEEE definisce infatti per ogni stato, cosa fa l'attività, quali sono gli input, quali sono gli output, quali sono i controlli e quali sono le metriche, ovvero ciò che vado a misurare durante l'attività. Le fasi sono:

- **problem identification:** prende in input la **modification request** e produce in output la **validated modification request** e la **process determination**, ovvero il tipo di **modello di processo che si andrà ad utilizzare**. Il modello di processo sarà scelto **sulla base del tipo di manutenzione da utilizzare**. Come controlli, La modification request deve essere **univocamente riconosciuta** e la modification request deve essere **inserita nelle repository**. In questa fase possono essere misurate il numero di MR sottomesse, il numero di MR duplicate o il tempo speso per la problem validation. In questa fase è necessario **stimare le risorse necessarie per cambiare il sistema** ed inserire la MR in un batch di cambiamenti schedulati per l'implementazione.
- **analysis:** si prende l'output della problem identification e lo si utilizza come input di questa fase, aggiungendo ad essi **documenti del progetto e informazioni sulle repositories**. Si produce il feasibility report, detailed analysis report, preliminary modification report, test strategy ed implementation plan. In questa fase **viene effettuata una feasibility analysis, andando a vedere le possibili soluzioni, alternative. Per ogni alternativa si vanno a valutare i costi**, determinando anche i benefici del cambiamento. Scelta una delle possibili alternative, **eseguo per quest'ultima detailed analysis**, individuo i requisiti che devono essere modificati, software components coinvolti e definisco una **strategia di testing** e piano di implementazione. Eseguo dei controlli afferenti alla technical review, verifico che la documentazione sia aggiornata, verifico la test strategy e issues legate a safety e sicurezza, producendo delle metriche come requirements changes, documentation error rates, effort per function area... **Queste sono metriche che vanno ad aggregare lo sforzo per vedere su quali parti del sistema vado ad intervenire più spesso**, potendo sapere quanto mi costa effettuare delle modifiche.
- **design:** prende i documenti della fase di analisi, il source code database, i documenti di progetto e produce **updated design baseline** (modificando il design si ha una nuova baseline di design), updated test plan, revised detail analysis, verified requirements,

revises implementation plan e documented constraints and risks. Le attività della fase di design sono quelle di **individuare le componenti software influenzate**. Mentre nell'impact analysis ci basiamo su alcune informazioni per capire se un determinato componente è impattato o meno, **quando faccio la progettazione individuo coloro che sono effettivamente impattanti**. Nella fase di **impact analysis** quindi si capisce se un certo componente potrebbe essere impattato ma non ho un livello di dettaglio tale da capire se un componente è sicuramente impattato. Implementando la modifica, per cui anche già in fase di design, si riesce a capire se delle componenti sono impattanti o meno, riuscendo inoltre a definire nuove componenti che non avevo stimato essere impattanti ma che invece verranno impattate. In questa fase si iniziano ad avere **metriche di prodotto**. Si documentano inoltre i cambiamenti, si crea un a test suite per il design e si selezionano i casi di test per il regression testing

- **implementation:** prende in input non solo l'output della fase di design ma anche documenti di progetto e di sistema. Si produce un software aggiornato, documenti aggiornati e updated training materials in quanto la modifica potrebbe avere un impatto sull'utente. Come output di questa fase si prevede anche lo **statement of risks**. Le attività svolte in questa fase sono la **scrittura di nuovo codice e il performing di unit testing, integrazione del codice modificato, condurre integration e regression testing, effettuare delle risk analysis e verificare la tracciabilità del design sul codice** per una lettura più facilitata del testing. Verrà inoltre condotto software inspection, ovvero una verifica statica del codice.
- **system test:** prende in input l'output della fase precedente e il report della test readiness review, producendo dei test report. In questa fase **i test sono effettuati su tutto il sistema, assicurandosi che il sistema modificato sia in linea con i requisiti originali e con le modifiche**.
- **acceptance test:** si produce un output funzionale. **Viene fatto un audit del sistema in modo tale da poter produrre una nuova baseline**. Dai test precedenti ed il test di accettazione si ha il sistema pienamente integrato potendo produrre la nuova baseline e l'acceptance test report. Facendo manutenzione correttiva non si farà un'acceptance testing, che verrà però fatta in caso di enhancement
- **delivery:** il sistema è rilasciato ai costumers per l'installazione e l'operazione

Seppur gli output di una fase vengano usati come input di un'altra, **non si va a specificare se il modello sia iterativo o incrementale** ma solo gli input necessari di ogni fase ed il loro output. Non tutti gli output devono essere prodotti da ogni intervento di manutenzione. **Gli output infatti, dipendono dal tipo di manutenzione che si va a fare sul sistema.**

Ogni attività ha associati 5 attributi:

- **activity definition:** processo di implementazione dell'attività
- **input:** input dell'attività
- **output:** output dell'attività
- **control: items che provvedono al controllo dell'attività**
- **metrics: items che vengono misurate** durante lo svolgimento dell'attività

4.7.2 ISO/IEC 14764

Lo standard 14764 descrive la maintenance usando gli stessi concetti dello IEEE/EIA 1219 ma **espressi in maniera differente**. Si avranno **diverse attività e ogni attività sarà composta da tasks**. Anche in questo caso il processo reagisce anche qui ad una change request e la modifica viene riportata anche sulla documentazione. Le 6 fasi sono:

- **process implementation:** sviluppa i piani di procedura della manutenzione. I task di questa fase sono la **definizione di un maintenance plan**, le MR/CR (modification request o change requests) ed il **configuration management**. Si avranno inoltre **step aggiuntivi** a quelli che vengono inseriti in questa fase. Effettuare un processo di migrazione ad esempio, non è semplice quanto eseguire un'operazione di manutenzione, **per cui è necessario fornire informazioni aggiuntive nelle varie fasi del ciclo**. È necessario inoltre definire un configuration management plan, in cui si va a definire ad esempio quali siano le componenti che di volta in volta devono essere cambiate per passare da un vecchio sistema ad uno nuovo. Se sto mettendo in piedi un processo di manutenzione correttiva, alla fase successiva non arriverà una modification request, per cui gli input ed output di ogni fase saranno dipendenti dal tipo di manutenzione
- **problem and modification analysis:** posso avere o un problem report o una modification request. Nel caso di **manutenzione correttiva ho un problem report**. Lo standard IEEE/EIA 1219 non prevedeva questo tipo di report, per cui **si aveva una modification request per qualsiasi tipo di problema**. Se c'è una failure però l'utente non mi richiede

una modification request, ma apre un ticket, quindi un problem report. Nei processi di enhancement ci può essere quindi una MR generale e poi delle MR che vanno a specificare il modo in cui la manutenzione è pensata, ovvero se incrementale, iterativa...

Su ogni pezzo che andrò a modificare avrò una richiesta differente per la modifica del pezzo. In questa fase quindi effettuo una **problem replication or verification**, in modo tale da **replicare il problema e capire da cosa** derivi lo stesso, **sviluppare poi delle opzioni per implementare la modifica** (in caso ci siano diverse alternative tra cui scegliere), **documentare i risultati e approvare la modifica effettuata**. Si avranno inoltre processi di controllo, che vanno a controllare le attività che sto svolgendo, ed i processi di supporto, così come previsti dallo standard ISO/IEC 12206. Il processo di problem resolution è il processo che sovraintende la manutenzione. Il **problem resolution process** nasce quando arriva una richiesta e termina quando il ticket afferente alla richiesta viene chiuso. Tale processo **potrebbe innescare un processo di manutenzione**, per cui il ticket diventa il problem report che chi fa manutenzione deve andare ad analizzare. Lo stesso vale per il processo di configuration management, processo separato, come il precedente, che si incrocia però con il processo di manutenzione. Come output, **questa fase produce una serie di documenti afferenti a come la modifica potrebbe impattare sul sistema**, il costo di tale modifica, determinare vincoli hardware e software che potrebbero risultare dalla modifica proposta, stimare i benefici della modifica... I principali task di questa fase sono:

- **modification request analysis:** si analizza la modification request
- **verification**
- **option task:** mi indica **cosa posso fare rispetto tutte le possibili alternative di soluzione**
- **documentation task:** mi dice **cosa devo fare per documentare le modifiche**. Se la documentazione non esiste è necessario sviluppare una nuova documentazione
- **approval:**
- **modification implementation:** in base a ciò che è stato definito dalla process implementation, **si esegue il processo di sviluppo per implementare le modifiche**. Si chiama processo di sviluppo poichè si suppone che la maggiorparte degli interventi non siano correttivi ma di enhancement, basandosi però su cose che esistono già all'interno del sistema. Si parte dall'architettura, dal MR record, la MR approvata, il source code, l'impact analysis report e dall'output della fase precedente e si produce codice modificato,

updated test plans, procedures, updated test reports... **Anche in questa fase si ha della quality assurance.** Non mi basta la quality assurance della fase precedente poichè **in questa fase si potrebbero avere delle metriche di prodotto che prima non avevo.** Nella fase precedente non avevo infatti il codice, disponibile in questa fase

- **maintenance review and acceptance:** similare a quello presente nello standard IEEE/EIA 1219. Ci assicuriamo che i cambiamenti effettuati al software siano corretti e siano fatti in accordo agli standard. Questa attività è inoltre supportata da processi di quality assurance, verification, validation... Ho in input il software modificato e come output produco la nuova baseline una volta che il software è stato accettato. **C'è meno interesse nel fornire tasks e steps, poichè molto dipende dal processo che vado ad implementare.** Le management reviews ci assicurano che le modifiche al sistema vengano fatte seguendo costi, tempo e schedule, mentre le peer review sono fatte internamente, sui documenti. In questa fase però ci sarà anche l'approvazione del cliente, ovvero la **review acceptance**, che consiste in operazioni di controllo sul software ma fatte esternamente al team
- **migration:** dovendo migrare un software **si parte dal vecchio sistema e si deve produrre un nuovo sistema.** Questa attività di migration però **non è l'attività che effettua la produzione della nuova baseline**, poichè la nuova baseline è in input a questa fase. Questa attività **fa quindi riferimento alla migrazione in esercizio dal vecchio sistema al nuovo sistema.** Le baseline in input a questa fase, vecchia e nuova, fanno entrambe riferimento allo stesso sistema che è stato reingenerizzato. Si sta **introducendo la versione migrata del sistema in esercizio.** Pianifico come effettuare la migrazione, **ma devo effettuare un'analisi**, intervenendo con un processo ben definito e di dettaglio sul come operare, **poichè i task** che vanno a definire le operazioni svolte nella fase di migration **non sono comunque pienamente sufficienti.** Si effettuano le operazioni di **parallel operations and training.** In questa fase poichè **non si ritira mai senza avere un nuovo sistema.** La migrazione dei dati viene fatta in parallelo in quanto non si può smettere di utilizzare un sistema ed utilizzarne un altro istantaneamente, sia a causa di **errori ai dati** che potrebbero occorrere, ma anche a causa di **errori che potrebbero verificarsi al nuovo sistema.** In tal caso si avrebbe infatti ancora il vecchio sistema attivo.
- **retirement:** a differenza della migration, **si ha un nuovo prodotto software**, per cui si sostituisce il sistema vecchio con un nuovo sistema, in cui al massimo ho solo

il problema della migrazione dei dati. **Anche in questa fase verranno eseguite le operazioni in parallelo**, poichè anche in questo caso potrebbe essere necessaria la migrazione dei dati. Nel **retirement c'è anche un replacement del sistema vecchio con un nuovo sistema** che potrebbe essere differente dal sistema precedente, **non basato sul vecchio**, mentre nella **migration il nuovo sistema è basato sul vecchio**

Il ciclo termina con la **maintenance and review acceptance**, per cui la messa in esercizio del software viene effettuata con la migration. **Man mano che effettuo la migrazione avrò dati sul vecchio e nuovo sistema**. Se effettuo una modifica sui dati del nuovo sistema durante l'operazione di migrazione, **la modifica deve essere fatta anche sul sistema vecchio**, mentre nel caso in cui i dati modificati non siano ancora sul nuovo sistema, **la modifica dovrà essere fatta solo sul vecchio sistema**. Per ogni componente che andrà a modificare **avrò la richiesta di modifica per quel singolo componente**. La richiesta generale mi permette di capire come organizzare il processo, mentre per la richiesta specifica si andranno ad eseguire le fasi di **problem modification analysis, maintenance review/acceptance e modification implementation**.

Le metriche utilizzate per valutare la bontà della manutenzione sono implicite, in quanto **lo standard IOS/IEC 14764 non si focalizza su quali metriche utilizzare, a differenza dello standard 1219**. Si focalizza però sui processi di supporto e di controllo, mentre lo standard 1219 si focalizzava solo sui processi di controllo.

CAPITOLO 5

Ciclo di vita del software

5.1 Introduzione: Software Configuration Management

La software configuration management (SCM) è **necessaria per consentire lo sviluppo concorrente, supportare software con versioni multiple in esercizio**, per capire ogni configurazione del sistema software da quali versioni di quali componenti è formata. Gli obiettivi sono quindi:

- **identificare univocamente le versioni di ogni componente in vari punti del tempo**
- **mantenere versioni di componenti precedenti.** In caso si abbia un difetto infatti è necessario vedere in quale versione della componente tale errore sia stato introdotto
- **mantenere la tracciabilità e l'integrità dei cambiamenti al sistema**
- **fornire una verifica per tutte le modifiche eseguite**

All'interno di un software, nel caso in cui si verifichi un problema in una determinata versione, non è detto che il problema risieda nella versione precedente, per cui è **necessario mantenere tracciabilità delle versioni del software e non solo l'ultima versione della stessa**. Software configuration management è una **project function**, ovvero attività che vengono svolte durante tutto il progetto, con l'obiettivo di effettuare attività tecniche e manageriali in modo più efficiente. Essa è composta da:

- **configuration item identification:** importante definire il nome degli schema e i modelli di feature per aggregare configuration items
- **promotion management:** definisce delle policy per andare a decidere quando posso fare un commit, ovvero quando posso mettere a disposizione degli altri sviluppatori ciò che ho prodotto
- **release management:** policy per rilasciare il sistema
- **branch management:** come faccio a gestire i branch
- **variant management:** come gestisco le varianti del software. Avendo delle versioni in esercizio che coesistono **ho bisogno di fare manutenzione e enhancement di tutte queste versioni.** Esse avranno parte di codice in comune e parte di codice non in comune. Se uso un modello basato sulle features, potrò capire quali sono le versioni in comune e quali no. Modelli per gestire le varianti potrebbero essere:



– **single project:** si applica quando il codice condiviso è molto. Si avrà chi lavora sul codice condiviso e chi si occupa delle parti afferenti alle varianti specifiche. I team saranno in numero minore o composti da meno persone. Il team più grande lavorerà sulla parte condivisa. Ci sarà uno sforzo di coordinamento ma sarà relativo, in quanto il grosso del software è condiviso.

– **redundant teams:** non mi importa di quale sia la parte condivisa e **ogni team si gestisce una propria variante.** Quando dovrà essere fatta una modifica di una feature condivisa, tutti i team dovranno andare a fare la modifica, per cui il codice si duplica. La parte condivisa di codice nel tempo sarà sempre minore poichè non si avrà coordinazione. Ciò si applica quando è poco il codice condiviso, altrimenti il costo del coordinamento diventa maggiore del beneficio

Il costo della comunicazione è sopportabile se la parte condivisa è prevalente, per cui si riescono ad abbattere i costi che invece avrei duplicando gli sviluppatori.

5.2 SCM Spectrum of Functionality

Le funzionalità offerte da CSM sono state classificate in 3 macro-aree:

- **product:** suddiviso a sua volta in identification, version control e system models and selection.

- **tool:** workspace control e building
- **process:** change management, status accounting, auditing

Il controllo dei cambiamenti fa parte dei processi, per cui **change management** è stato inserito nella macroarea del processo.

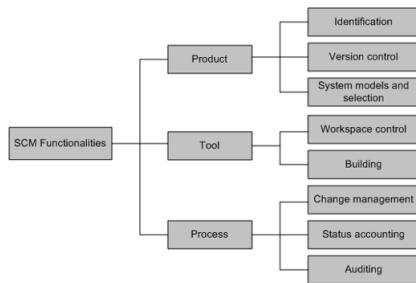


Figura 5.1: SCM funcionalities

5.3 Product

5.3.1 Identification

Per prima cosa bisogna **individuare gli elementi da porre sotto configuration management**. L'idea è mettere sotto configuration management qualsiasi artefatto prodotto durante il **ciclo di vita del software**, anche se si predilige il codice. Mettere sotto configuration management significa non solo **fare versioning**. I requisiti ed i casi d'uso sono estremamente volatili per cui **ogni cambiamento che faccio ad un requisito durante la fase di requirement analysis può avere un impatto su altri requisiti** e dato il fatto che il processo di requirement engineering è molto variabile, ovvero le cose possono cambiare molto velocemente, **è inutile andare ad introdurre un processo di change management durante questa fase**. Introducendo un processo di change management **si andrebbe di molto a rallentare la fase di analisi dei requisiti**, per cui non si introduce il change management fin quando **non si produce la baseline per l'intero documento di analisi dei requisiti**. Mettere sotto configuration management vuol dire che **ad ogni cambiamento della baseline si necessita di una richiesta formale di cambiamento**

5.3.2 Version control

Ci consente di **essere sicuri di star lavorando sotto una determinata versione del prodotto software**. Deve esserci una funzionalità all'interno della software configuration mana-

gement di version control che **non solo va ad interpretare gli artefatti come configuration items**, ovvero come **elementi per cui produrre diverse versioni, ma anche individuare relazioni tra gli i configuration items**. Senza queste relazioni **non è possibile fare la build del sistema**. Anche l'executable è un configuration item. L'idea del version control è di avere due file separati: **master copies e working copies**. Si scarica dalla master directory i files e si lavora con le copie dei files, fin quando quest'ultimi non vengono reinseriti nella master directory. Gli sviluppatori fanno il **checkout delle working copies** dalle repositories e fanno poi il **checkin delle working copies dopo aver effettuato operazioni su di essi**. Non si può modificare il file sulla master directory quindi è necessario fare un **checkout per ottenere una copia locale** su cui posso operare. **Se io sto lavorando sul file master in locale altri developers non potranno fare il checkout**. Facendo il **checkin** la copia locale che ho modificato sostituirà la copia master, per cui **tutti vedranno la modifica effettuata**.

Se tutto il codice si trova su una singola macchina, era necessario fare telnet, andare sulla directory, lavorare all'interno del terminale remoto. Erano quindi necessarie le credenziali per accedere la macchina e i permessi per lavorare sulla macchina. Si è poi passati a sistemi concorrenti, come CVS che consente di scaricare su un'altra macchina le versioni potendo avere concorrenza su uno stesso file. Si potrebbero avere branch paralleli con cui si lavorare insieme al main trunk. In fase di manutenzione potrebbe essere normale avere più richieste di modifica che vanno ad impattare sulla stessa classe. Il **main trunk contiene il codice del sistema in esercizio su cui vengono fatti interventi correttivi ma non interventi di enhancement pesanti**.

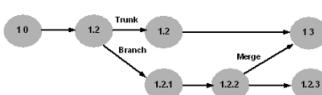


Figura 5.2: Version control

5.3.3 System models and selections

Questa fase permette quindi di **avere un accesso selettivo a parti di versioni di artefatti aggregati**. È inefficiente gestire un progetto file-by-file, avendo quindi necessità di supportare artefatti aggregati, in modo che le persone che lavorano alla manutenzione possono forzare la consistenza.

Se ho un aggregato, un configuration item, **esso può essere o un singolo item o un configuration item aggregated**. Nel caso sia un configuration item aggregated per capire da

cosa sia composto è necessario definire un **modello di relazioni** in modo tale da poter poi **selezionare tutti gli elementi che mi consentono di costruire il configuration management aggregated**. Data una feature che per essere modificata necessita la modifica di diversi files, **è necessario che il sistema sia in grado di dire quali files debbano essere modificati**. Per evitare inconsistenti si usano dei modelli per ci permettono definire quali siano i files che devono essere modificati. Per esempio, leggendo il makefile so quali sono i files necessari per comporre il file e quindi quali devo andare a modificare.

5.4 Tool

5.4.1 Workspace control

Ambiente che consente al manutentore di **fare building, costruire e testare i cambiamenti in ambiente isolato**. In un sistema di configuration management le **versioni software sono salvate in repositories che non possono essere accedute direttamente**. Quando devo modificarle devo **operare su una copia locale**. Un workspace può essere semplice come la home directory di un programmatore o un meccanismo complesso integrato all'interno di un ambiente integrato. le funzioni eseguite in un workspace sono:

- **sandbox**: il checkout dei file, **scarica i file dalla master directory al workspace locale**. Se usiamo un **sistema di versioning concorrente** non abbiamo bisogno di bloccare il **file originale**
- **building**: generalmente **memorizza le differenze tra le versioni successive** per salvare **spazio**, memorizzando quindi i delta. **Partendo dai delta e dalla versione base posso costruire le versioni successive**. Facendo il building, il sistema deve **costruire i source code file partendo dai delta**, si creano i files e poi si espandono mediante i delta
- **isolation**: ogni sviluppatore mantiene un proprio workspace, per cui lo **sviluppatore fa le modifiche al codice sorgente**, compila i files, esegue i test e fa il debug del codice **senza impattare quindi il lavoro di altri sviluppatori**.

Grazie alle relazioni di feature model io faccio una build dell'aggregato. Facendo però la build, ogni file sorgente deve essere costruito a partire dai delta.

5.4.2 Building

Devo costruire un file eseguibile partendo dalle versioni dei file sorgente. Devo, inoltre, fare la build delle versioni precedenti, per cui anche il configuration management aggregate deve essere sotto versioning. Se voglio fare la build di una versione vecchia, devo avere istanze di relazioni e non solo relazioni. Devo sapere ogni **versione di ogni configuration item**, a **quali versioni fa riferimento nei file sorgente**. Sarà necessario sapere, per **ogni configuration management aggregate da quali versioni dei file sorgenti dipende**. Il **makefile esplicata solo le relazioni tra i files** ma non gestisce questo aspetto, mentre un **sistema di building deve gestire anche tale aspetto**. La build sarà poi necessaria per convalidare il prodotto.

5.5 Process

5.5.1 change management

Processo più importante poichè è il **processo tramite il quale si decise se andare ad apportare modifiche al sistema partendo dalla change request**. Esso è un processo che si va ad incastrare con il processo di manutenzione, **partendo dalla change request, modification request e poi si va a valutare l'impatto**.

Al processo di change management arriva la **change request**, potendo fare una scelta preliminare. Se si passa all'analisi allora **abbiamo fatto partire un processo di manutenzione**. Al termine dell'**impact analysis** si torna nel processo di **change management** e i risultati dell'impact analysis vengono valutati dal **change control board che decide se effettuare o meno la modifica**, sulla base dell'analisi costi/benefici. Il processo di manutenzione è un processo composto da attività fatte dalle persone, mentre il **sistema di configuration management supporta il processo di manutenzione**, fornendo delle funzionalità per memorizzare tutte le informazioni. Per esempio anche un processo di anomalies management potrebbe innescare un processo di software maintenance in cui devo effettuare delle modifiche al sistema. Solitamente si usa un **issues tracker per tener traccia delle change requests aperte e chiuse**. Si avrà un link tra le change requests, quindi issues, e i files che sono stati coinvolti in quella change request.

5.5.2 Accounting

Serve a mantenere record formali di configurazioni dei sistemi. Produce **report periodici sullo status delle configurazioni**. Descrive il **prodotto usato per verificare le configurazioni**

del sistema e mantiene la storia delle change request. La storia della feature andrà ad indicare, versione per versione **quali sono i files interessati e cosa è stato fatto nel passaggio da una versione all'altra.** Tra due release, infatti, potrebbero esserci differenti versioni dei file. La **storia dei cambiamenti** andrà ad indicare **quali cambiamenti sono stati fatti**, quando sono stati fatti, chi li ha fatti.

5.5.3 Auditing

Processo che serve per **approvare le baseline**. La **software review** viene fatta per decidere **se una nuova baseline può essere prodotta o meno**. È necessario poter **fare roll-back ad una versione precedente stabile**, nel caso la nuova baseline non venga approvata. I tipi di auditing che possono essere fatti sono:

- **audit for functional configuration:** verificare e validare se il **software soddisfa le specifiche software**. Effettua una **verifica sul funzionamento del sistema**. La verifica può essere fatta tramite testing, tramite esecuzione del sistema
- **audit for physical configuration:** verifica se i documenti rappresentano accuratamente **il software**. Esso rappresenta la **correttezza dei documenti rispetto al software**. La verifica che può essere fatta è statica

5.6 SCM Process

Ci sono 3 attività fondamentali:

- **pianificazione:** definisco i processi, individuo gli stackholders, sviluppo o introduco i tool per fare software configuration management. Avendo messo in piedi il sistema di configuration management, con tutte le procedure ed i strumenti necessari, a questo punto è possibile passare alla fase successiva. Non tutti i cambiamenti sono visti dalla **configuration control board**, ovvero gli **stackholders**. **Piccoli gruppi rivedono e approvano** la maggior parte dei cambiamenti. Quando ci si rende conto che è necessario l'intervento di qualcuno che sta più alto in grado si passa al change control board. Sarà comunque necessario individuare i gruppi che approvano richieste minori e chi è il **change control board che approva richieste maggiori**. Passo successivo è l'identificazione degli items
- **baseline development:** identifico gli item da controllare, identifico le baseline e sviluppo lo schema per gli identificatori. Non devo quindi individuare solo le **con-**

figurazioni, ma anche gli **aggregati che costituiscono le baseline**. Sviluppo infine il rating schema per gestire files e aggregati

- **configuration control:** valuto i cambiamenti, li approvo o meno, tengo traccia dei cambiamenti fino alla chiusura della change request, necessitando quindi di sapere a chi è stato assegnata, devo poi aggiornare la baseline e pubblicare i report. Una volta aggiornata la baseline, la baseline deve essere approvata, per cui si ha una **fase di audit**

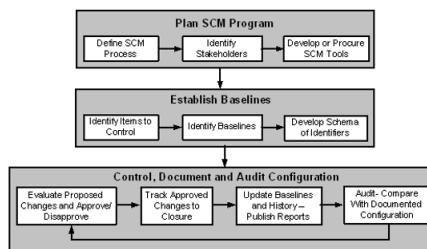


Figura 5.3: SCM process

5.7 Change request flow

Una volta che una **CR è sottomessa**, essa viene **visionata**. Per individuare se una CR è uguale o simile ad un'altra CR si possono utilizzare **tecniche di information retrieval**, che permette di capire la somiglianza testuale fra una CR e le altre. Nel caso ad esempio in cui una CR sia simile ad altre CR, sarebbe possibile, magari, **assegnare la CR alla stessa persona che ha gestito le altre similari**. Attualmente, **tecniche di text processing sono molto utilizzate per gestire questo tipo di operazioni**, in modo più o meno automatico. Se sono in un closed source system, sapendo che ognuno si occupa di una determinata parte è semplice assegnare la CR, ma, trovandomi in una community open source, è più complesso assegnare le CR.

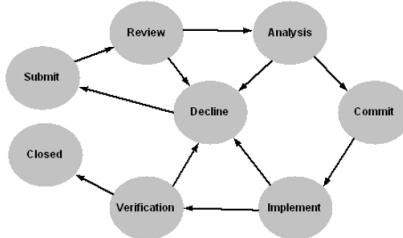


Figura 5.4: Change request flow

Nella fase di **commit** del life-cycle si decide di integrare la modifica. Se mi accorgo ad esempio durante l'implementazione che l'imapct analysis non era stata eseguita bene per cui i danni prodotti dall'inserimento della feature potrebbero essere elevati, è possibile fermare l'**aggiunta della feature**. Nel caso in cui nella **prima review** si va ad identificare la CR come copia o identica ad un'altra, essa può essere declinata. Nella fase di **implementation** è intrinseca la **fase di design e testing**, per cui nella fase di **verification** si va ad effettuare solo l'**audit**, rappresentando la fase di acceptance. è possibile declinare la CR in qualsiasi fase, tranne in quella di **commit**. Alla fine dell'**audit** si decide se implementare o meno la **baseline**.

5.8 Baseline

Una baseline è una **specificità del prodotto** che è stata formalmente rivista ed approvata dal management responsabile e che serve come **base per futuri development**. La baseline farà da base per il nuovo codice e può essere cambiata solo attraverso una procedura di controllo formale. Le baseline vengono suddivise in:

- **development baseline**: documenti che vengono prodotti durante il ciclo di vita e che vengono approvati formalmente. Servono a coordinare le attività e a valutare il progresso delle attività
- **functional baseline**: baseline che servono ad ottenere una prima esperienza da parte dell'utente
- **product baseline**: release che vado a rilasciare. Servono a coordinare le vendite e supportare il cliente.

Gli schemi possono essere di vario tipo. Lo sviluppatore che lavora nella sua working area può creare delle versioni che metterà a disposizione nella master directory che può essere controllata ed i cambiamenti devono essere autorizzati. Nella **software repository**, chiamata anche **static library** poichè non avviene nessun cambiamento, verranno invece messe a disposizione le relative **release**. Posso condurre baseline di diversi aggregati. Il termine **versione** viene usato quando si ha un **enhancement**, mentre **revision** viene usato quando c'è una **correzione**.

CAPITOLO 6

Reengineering

6.1 Introduzione: Reengineering

Il reengineering è l'**esame, analisi e ristrutturazione di un software esistente per ricostruirlo in una nuova forma e la susseguente implementazione della nuova forma**. Gli obiettivi sono quelli di comprendere il software esistente, specifica, design, implementazione, documentazione e migliorare le funzionalità e la qualità degli attributi del software. Se migliorano le funzionalità, il sistema che ho realizzato non deve necessariamente avere le stesse funzionalità del sistema di partenza. Si potrebbero quindi modificare le funzionalità del sistema.

Spesso si fa reengineering per **migliorare la manutenibilità**, per preparare il software ad un miglioramento funzionale o per migrare il software verso una nuova tecnologia. In tal caso infatti si potrebbero aggiungere funzionalità in quanto, se si migra un'applicazione da stand-alone a client-server, è chiaro che si possano modificare le funzionalità. La **migrazione verso una nuova piattaforma è propedeutica all'aggiunta di funzionalità**.

6.2 Concetti del reengineering

I concetti fondamentali usati durante lo sviluppo sono **abstraction** e **refinement**. Essi sono **concetti utilizzati durante lo sviluppo e che servono maggiormente durante la fase**

di reengineering. Essi sono importanti poichè bisogna **comprendere il software esistente**, quindi serve fare astrazione, passando a livelli di astrazione maggiore.

In alcuni contesti per il reverse engineering si può partire dal codice binario, anche se normalmente si parte dal source code. Il **costo del reengineering** = reverse engineering + delta + forward engineering, dove delta è il **costo per catturare le alterazioni fatte al sistema originale**.

Il **refinement è importante**. A differenza del forward engineering in cui parto senza codice, per cui il passo di refinement è assente, nel caso del reengineering, **avendo il codice**, quest'ultimo è **utilizzabile nel passo di refinement per riorganizzarlo in maniera diversa** in accordo con cosa mi dice l'architettura.

6.2.1 Astrazione

Attraverso il processo di astrazione, ovvero sto **partendo da un livello di astrazione più basso**, ed applicando un livello di astrazione più alto (sto facendo reverse engineering) **vado a produrre una lista che si focalizza solo su alcune caratteristiche del sistema**, nascondendo quindi gli aspetti non interessanti. **Facendo astrazione quindi nascondo le informazioni che non mi servono**, informazioni che si trovano a livello più basso

6.2.2 Raffinamento

Con il raffinamento invece, il **livello di astrazione della rappresentazione del sistema viene gradualmente decrementato**, per cui **riduco il livello di astrazione** e vado ad un livello più basso, **avendo quindi un numero maggiore di dettagli**.

6.2.3 Forward e reverse engineering

Il **forward engineering** è quello che facciamo quando sviluppiamo un sistema software. Il processo inverso, che si trova alla base dell'astrazione è il **reverse engineering**. Esso permette di analizzare il software e le sue componenti e di rappresentare il sistema ad un **livello di astrazione più alto o in altre forme**.

Esempio: Prendendo il codice sorgente e tramite parsing tiro fuori un abstract syntax tree, eventualmente arricchito con informazioni semantiche o con informazioni sul flusso di controllo si ha una rappresentazione non ad un livello di astrazione differente ma allo stesso livello in una forma differente.

6.2.4 Livelli di astrazione

I concetti dell’astrazione sono utilizzati per creare 4 livelli di astrazione: **concettuale**, **requirements** (specifiche dei requisiti), **design** ed **implementation**. Anche ciò che viene prodotto, viene prodotto a 4 livelli di astrazione differenti.

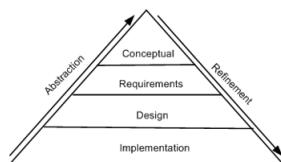


Figura 6.1: Livelli di astrazione

Con il processo di **refinement** si parte dal livello **concettuale** e si arriva al livello **implementativo**, mentre il processo di **abstraction** segue la strada opposta, partendo dall’implementazione e arrivando al concettuale. Fondamentalmente, nei processi di astrazione possiamo partire da un qualunque livello di astrazione e ci fermiamo ad un qualunque livello di astrazione.



6.3 Principio di alterazione

Un ulteriore principio è quello dell’**alteration**. Facendo abstraction e refinement **senza cambiare nulla**, quello che ottengo alla fine del refinement è quello che avevo all’inizio, o almeno in teoria. In pratica **ogni volta che prendo una decisione sul design o sull’implementazione, potrei decidere di implementare la cosa in maniera diversa**, facendo scelte diverse.

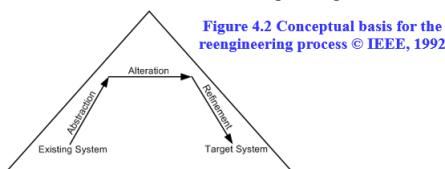


Figura 6.2: Principio di alterazione

I **cambiamenti apportati al sistema fanno parte dell’alterazione**. Posso arrivare allo stesso codice tramite strade diverse e dallo stesso codice posso arrivare a strade diverse. **Senza una fase di alterazione**, eseguendo solo astrazione e refinement, **non è detto che io ottenga lo stesso codice, ma otterrò fondamentalmente codice che farà le stesse cose**, poiché non si stanno cambiando i requisiti. Con il **principio di alterazione**, **si decide il livello a cui si**

vuole fare reengineering, facendo astrazione fino a quel livello. Si fa poi una **ristrutturazione**, un'**alterazione della rappresentazione** di quel livello e si fa poi un'**operazione di refinement**.

Posso fare quindi **reengineering a diversi livelli di astrazione**, come **re-think a livello concettuale**, **re-specify al livello di requirements**, **re-design a livello design** e **re-code a livello implementativo**. Lo stesso concetto può essere espresso come un modello a ferro di cavallo. Si parla prima del reengineering architetturale, poi una trasformazione dell'architettura e poi si va a sviluppare la nuova architettura.

Esempio: Volendo passare un programma da un linguaggio all'altro, potrei fare un rewrite line to line, oppure, potrei trattare il problema ad un livello più alto usando poi quel livello come base del refinement.

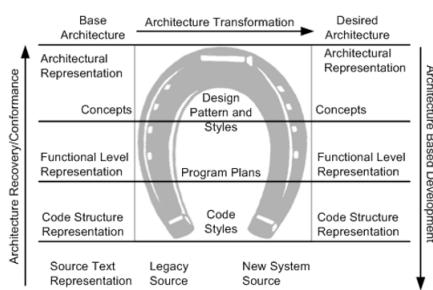


Figura 6.3: Modello a ferro di cavallo

6.4 Rehosting

Un altro termine usato per il reengineering è il **rehosting**. Il rehosting indica un **reengineering del codice senza aggiunta o riduzione delle features** nel source code target, non avendo quindi un reengineering a livello di target, andando solo a migliorare la qualità.

6.5 Tipi di cambiamenti del reengineering

In base alle modifiche richieste, le caratteristiche del sistema sono divise in:

- **recode:** **cambio le caratteristiche del programma**. Posso avere **attività di migrazione** verso una nuova tecnologia, **attività di normalizzazione, ottimizzazione e refactoring**... Per questo tipo di attività è comunque necessario tirare fuori dal codice una **rappresentazione diversa**. È quindi **necessario costruire un control flow graph**, che non mi dà informazioni aggiuntive rispetto al codice ma è **manipolabile**, ed essendo un grafo posso applicare algoritmi su di esso

Esempio: scrivere il programma in un altro linguaggio o eliminare i commenti.

- **redesign:** vado a **rivedere l'architettura del sistema**. Modifiche comuni sono: **ristrutturazione dell'architettura, modifica, rimpiazzare un algoritmo o una procedura con una più efficiente**.

Esempio: La struttura dei files può essere più o meno aderente ad uno schema relazionale. Più è aderente ad uno schema relazione più è semplice migrarlo ed effettuare reengineering in un database relazionale

- **respecify:** **cambio le caratteristiche dei requisiti**. Posso **cambiarli nella forma o nello scope** dei requisiti
- **rethink:** si vanno a **manipolare i concetti**, parlando a livello non funzionale. Vado quindi a **modificare le caratteristiche concettuali del sistema**. Viene eseguito quando cambia la **mission operativa del software**, ovvero quando **si deve rivedere completamente il motivo per cui si usa tale sistema**

Esempio: passaggio dallo sviluppo di un telefono cellulare normale ad uno smartphone dovendo quindi trasformare il software da un software per telefono cellulare ad un software per uno smartphone

6.6 Rewrite

La strategia **riflette il principio dell'alterazione**. Attraverso l'alterazione il **sistema viene trasformato in un nuovo sistema preservando il livello di astrazione** del sistema originale. La **rewrite si occupa solo dell'alterazione non usando nessuna forma di rappresentazione diversa**, ovvero **modifica del codice line by line**. **Non posso effettuare il rewrite passando da un linguaggio all'altro**, almeno che il paradigma procedurale non rimanga lo stesso

6.7 Rework

Si **occupa dell'astrazione, alterazione e refinement**. Nel refinement, dopo aver fatto i rework, posso utilizzare codice legato agli elementi dell'architettura.

6.8 Replace

è possibile eseguire solo **astrazione e refinement**. Con il replace non si possono mai modificare le funzionalità, non potendo fare reengineering a livello di requirements o a livello di re-thinking. Il sistema viene costruito ad alto livello e viene poi implementato. Facevo ciò che faccio con lo sviluppo solo che l'astrazione non mi viene dai requisiti utente ma dal codice. Se voglio implementare il sistema in una forma differente, **tiro fuori i requisiti dal sistema e lo reimplemento**.

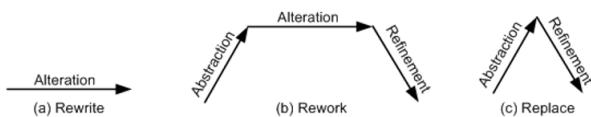


Figura 6.4: Rewrite, rework e replace

Esempio: Facendo reengineering devo astrarre le informazioni, capire determinate dipendenze, e ciò le capisco mediante gli algoritmi che vado ad utilizzare. Capisco che ad esempio un campo di un file è usato come chiave esterna di un altro file riconoscendo ad esempio un'operazione di join. Riconosciuta l'operazione posso andare ad apportare delle modifiche migliorative come la realizzazione di un join che crea direttamente un file dai due files.

6.9 Reengineering variations

La tabella mostra le operazioni che possono essere fatte rispettivamente per rewrite, rework e replace. **Yes:** può produrre un target system, **Yes***: come Yes ma il successo è minore, **No:** non si può iniziare all'abstraction level e **Bad:** Un target system può essere creato ma il risultato è quasi sempre pessimo

Starting Abstraction Level	Type Change	Reengineering Strategy		
		Rewrite	Rework	Replace
Implementation Level	Re-code	Yes	Yes	Yes
	Re-design	Bad	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Design Level	Re-code	No	No	No
	Re-design	Yes	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Requirement Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	Yes	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Conceptual Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	No	No	No
	Re-think	Yes	Yes*	Yes*

Figura 6.5: Rewrite, rework e replace

6.10 Reengineering Process

Un processo di reengineering è insieme ordinato di attività per effettuare specifici task. Ci sono diversi modelli di processo descritti in letteratura, ad esempio:

- **big bang:** dopo aver **effettuato il reengineering, si rilascia il sistema in un'unica volta.** è spesso utilizzato se il reengineering non può essere diviso in parti. è una **follia però voler sostituire il nuovo sistema con il vecchio tutto in un'unica volta.** I rischi di tale tecnica sono pari ai rischi legati allo sviluppo di un nuovo sistema. Questa tecnica è inoltre **molto costosa e lunga.** è come se si sviluppasse il sistema da 0, pur utilizzando parte del software esistente. Tale processo infatti sarebbe talmente lungo da non garantirmi che il software, alla fine, mi servirà ancora
- **approccio incrementale:** il **sistema è rilasciato in modo incrementale.** Con questa tecnica si hanno meno rischi, poichè **man mano si effettua il reengineering,** che diventa parte dell'evoluzione del sistema, **si controlla che non siano stati inseriti errori.** Come vantaggi si hanno la **semplicità nel controllare gli errori** ed è più semplice capire quali **siano i progressi.** Gli svantaggi sono che i **tempi potrebbero essere più lunghi** causa di integrazioni o controlli da fare ogni volta che viene integrato un nuovo pezzo. Nel caso in cui non possa essere fatto un reengineering di tipo incrementale, conviene piuttosto cercare altre soluzioni
- **approccio parziale:** mentre nell'incrementale io modifco tutto il sistema, **in questo caso vado a reenginerizzare solo una parte del sistema,** come reenginerizzare solo un'interfaccia utente... Il sistema viene inizialmente partizionato avendo da una **parte il sistema su cui effettuare reengineering** e dall'altra parte il sistema da non modificare. Viene poi effettuato il reengineering e viene successivamente integrata la parte su cui è stato applicato il reengineering con la parte non modificata. Per la parte da reenginerizzare è necessario capire se **applicare approccio incrementale o big bang.** Riduco lo scope dell'engineering quindi e **mi costa meno in termini di tempo e soldi**
- **approccio iterativo:** viene , la cui durata è breve. In questa fase ci sono quattro tipi di componenti: le **componenti vecchie non renginerizzate, le nuove renginerizzate, le vecchie renginerizzate e le nuove componenti messe in esercizio.** Con questo approccio il vantaggio è che **si garantiscono operazioni continue** e i mantainers e gli users acquistano familiarità con il sistema

- **evolutinary:** modo diverso di vedere l'approccio iterativo e incrementale. In questo caso le componenti sono raggruppate in base alle funzionalità e renginerizzate in nuove componenti



CAPITOLO 7

Code reverse engineering

7.1 Introduzione: Code reverse engineering

Il reverse engineering è un processo fondamentale che mi porta da un livello di astrazione ad un livello di astrazione più basso. Il reverse engineering non è un qualcosa che si fa esclusivamente per l'engineering ma si fa anche per altri motivi.

Quando si parla di reverse engineering si parla spesso di code reverse engineering. Anche se potrei applicare reverse engineering a qualsiasi livello, **quando si parla di reverse engineering si parte dal codice**.

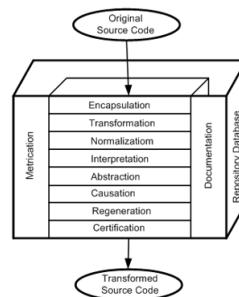


Figura 7.1: Code reverse engineering

Il termine di code reverse engineering viene dall'ingegneria elettrica dove si vedevano i circuiti e da essi si astraevano gli schemi. Esso è stato poi applicato al software, sia nel contesto del binary che per reverse engineering e da livelli più alti.

7.2 Definizione formale di code reverse engineering

Il code reverse engineering è una **pratica che consiste nel prendere un programma esistente e analizzarne il codice sorgente o il codice binario per comprendere come funziona internamente**. Esso è stato definito come un processo per **individuare delle componenti di un sistema** in esercizio, individuare le **relazioni tra le componenti e rappresentare il sistema a livelli di astrazione più alta o in un'altra forma**. Esso è principalmente usato per:

- **ridocumentare il sistema**, estraendo rappresentazioni diverse a livello più alto che possono ridокументare il sistema,
- a **livello di object design**, ovvero progettazione di basso livello. La possibilità di avere una **cross reference list**, che per ogni variabile definisce dove essa sia dichiarata... è **una rappresentazione a livello di codice**, che fornisce però dati per documentare il sistema e costruire un dizionario dei dati.
- Volendo salire a livelli più alti si può utilizzare il code reverse engineering anche per **effettuare design recovery**.

7.3 Design recovery e Code reverse engineering

Il grosso delle attività di reverse engineering è usato per fare **design recovery**. Questo processo **mira a comprendere la struttura e l'architettura del software**, catturando il **design concettuale e logico del sistema**. È molto difficile salire a livelli di astrazione più alti. Già a livello di design non è un qualcosa che può essere fatto automaticamente. **Man mano che si va avanti nella progettazione ci sono delle scelte che devono essere fatte**. Teoricamente si potrebbe arrivare fino alla fase concettuale, ma nella pratica **si riesce a tirare fuori dal codice, attraverso il design, l'architettura del sistema**. Volendo estrarre i requisiti **non si può partire dal codice mediante analisi statica**. Per ricostruire i casi d'uso di un sistema è **necessario usare il software**, vederlo in esecuzione, **riuscendo a tirar fuori degli scenari** da cui sono poi estratti i casi d'uso, piuttosto che i requisiti funzionali stessi.

Non è facile né automatico capire quali dettagli nascondere e quali dettagli possono essere interessanti ad un livello di astrazione più alto. Dal codice potrebbero essere estratti ad esempio dei sequence diagrams, che sarebbero però a livello di progettazione, contenenti molti dettagli. Sarebbero **differenti dai sequence diagrams** estratti a livello di analisi, che servivano solo ad **individuare gli oggetti del dominio applicativo e le interazioni tra oggetti e dominio applicativo**.

Non è quindi automatico capire quali dettagli nascondere e quali mantenere ad un livello di astrazione più alta. Il processo di code reverse engineering è **automatizzabile fino al design recovery**, poichè si tira fuori una descrizione del sistema e ciò può essere fatta mediante analisi del codici. Per arrivare ad un livello alto architetturale, però, è **necessario ricostruire delle decisioni che sono state fatte dal progettista**. Il progettista è **partito da un design di alto livello ed è passato ad un design di basso livello**, prendendo delle decisioni nel mentre. Poichè ci sono delle scelte non è automatico costruire il modello architetturale che aveva in mente lo sviluppatore. Tool automatici consentono di costruire livelli più bassi. Tutto ciò che è più in alto dal punto di vista architetturale ha bisogno dell'utente.

Nel **forward engineering** si parte dai **requisiti** e si arriva all'**implementazione**, mentre nel **reverse engineering** si parte dall'**implementazione** per costruire il **design** potendo fare inoltre **design recovery**.

Passando dall'**implementazione** al **design** è come se si dividessero le strade. Aggiungendo **informazioni** a quelle estratte dal codice, si ottiene una fase di **recovery design**, mentre non aggiungendo **informazioni** si ottiene **reverse engineering**. Per risalire ai requisiti si ha quindi bisogno di un intervento umano. È possibile estrarre **casi di test**, ma è necessario osservare il sistema, per cui non è una fase automatizzabile. La differenza quindi tra **design recovery** e **reverse engineering** è che il **design recovery** è meno **automatizzabile**. In ogni fase è possibile fare **restructuring**

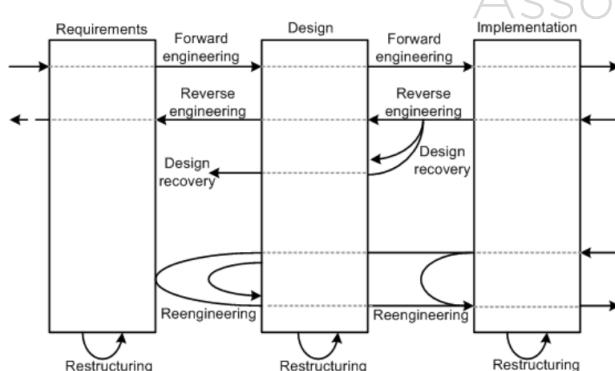


Figura 7.2: Code reverse engineering

7.4 Obiettivi del reverse engineering

Gli obiettivi del code revser engineering o reverse engineering sono:

- **generare viste alternative** rispetto al codice

- recuperare informazioni perse
- sintetizzare livelli di astrazione più alti
- individuare side effects: Si potrebbe fare reverse engineering per costruire rappresentazioni del codice che mi aiutino a fare impact analysis
- facilitare il riuso: Riuscendo ad individuare l'astrazione dietro al codice posso, sulla base delle specifiche che devo implementare, riusare componenti. Non avendo la specifica delle componenti non potrò riutilizzarle
- gestire la complessità: avendo livelli di astrazione più alti posso governare la complessità insita del codice

7.5 Standard e reverse engineering

Il reverse engineering è termine usato anche nella IEEE Standard per la Software Maintenance. I passi documentati dalla IEEE per lo standard della software maintenance sono:

- Partizionare il codice sorgente in unità
- Descrivere il significato delle unità e identificare le unità funzionali. Posso fare reverse engineering per tirare fuori la specifica di una funzione (precondizioni e postcondizioni). Questo studio può essere fatto mediante l'esecuzione simbolica. Essa consente di eseguire funzioni in maniera simbolica.
- Crea input ed output delle unità
- Descrivo le unità connesse
- Describe le system application e la struttura interna del sistema
- Crea una struttura interna del sistema

Le prime tre afferiscono a **local analysis**, poiché si fa l'analisi della singola funzione, mentre le altre 3 riguardano invece l'architettura, afferenti ai **global analysis**. C'è però da far notare che nel caso delle precondizioni, una local analysis potrebbe non bastare dovendo infatti analizzare anche le funzioni chiamanti. Sicuramente si può derivare dalla sola analisi della funzione la postcondizione, non è detto che possa essere ricavata la precondizione. Ciò che resta difficile individuare è l'invariante. Per la precondizione invece si va a controllare se

la funzione analizzata ha degli if prima delle operazioni. In caso non ci siano, si vanno a vedere il codice delle funzioni che chiamano la funzione e vedere lì se ci sono delle precondizioni.

Esempio: Data una funzione scritta in un linguaggio di programmazione procedurale, che ha come input ad esempio 3 valori, sostituiamo le 3 variabili con valori simbolici e andiamo a calcolare mediante esecuzione simbolica, i valori delle variabili che dipendono da esse. Ogni variabile all'interno del programma sarà espressa in funzione dei valori simbolici delle variabili in input.

7.6 Esecuzione simbolica

Invece di parlare di stato del programma si parla di **stato simbolico del programma**. Lo stato simbolico del programma sarà il mapping per ogni punto di esecuzione del programma delle variabili sulle espressioni simboliche interne dei valori simbolici di input che assumono le variabili in quel momento. Il problema nell'analisi del codice è che **un codice potrebbe dar vita a più percorsi** che si andrebbero sì ad unire alla fine, ma che effettueranno, molto probabilmente, operazioni differenti nel frattempo. Per ogni percorso sarà necessario definire **una path condition**, ovvero la **condizione che dovrà essere vera affinché quel cammino sia percorso**. I risultati prodotti dai percorsi verranno poi uniti per creare una condizione composta mediante OR. Tale condizione risultato indicherà la postcondizione, ricostruita quindi mediante esecuzione del codice. La semplificazione dovrebbe portare ad avere una postcondizione più comprensibile.

Nel caso sia presente un ciclo e non un if, è necessario calcolare mediante tecniche delle **invarianti di ciclo**, che indicano **quale sarà la condizione rispettata quando si esce dal ciclo**. Si parla di precondizioni, postcondizioni e poi stepwise refinement poichè si va a decomporre partendo dalla specifica in condizioni che ad ogni passo devono essere vere. Dovendo costruire la specifica, dovrei ricostruire l'invariante di ciclo, altrimenti si andrà in loop. **Qualunque sia il numero di volte che verrà eseguito il ciclo, l'invariante di ciclo sarà la medesima**, ricavabile in maniera più o meno automatica.

7.7 Reverse enginnering ed altri utilizzi

Il reverse engineering non serve solo per il reengineering ma serve anche per il **riuso, documentazione, design recovery** ed è inoltre usato per vedere se **esistono istanze di design pattern all'interno del codice**. Per capire se ci sia un'istanza del design pattern **si applica**

analisi statica, poichè in quanto design pattern strutturale basta il codice per capire se una determinata struttura sia presente o meno.

7.8 Paradigma Goal/Model/Tool

È stato definito un **paradigma per lo sviluppo di tool di reverse engineering** chiamato Goal/Model/Tool. Esso è un paradigma ad alto livello che **fornisce frameworks per usare metodi e tool disponibili ma anche per svilupparli** e per rendere il processo ripetitivo. Goal/Model/Tool si è **ispirato al Goal/Question/Metrics**. Goal/Model/Tool **partiziona il processo per il reverse engineering in tre fasi ordinate: Goals, Models e Tool**.

7.8.1 Goals

In questa fase **si ragiona per mettere in piedi il processo di reverse engineering**, analizzando le ragioni per cui bisogna fare reverse engineering. È **necessario capire quali sono le informazioni estratte dal codice e come esse possono essere rappresentate**. È necessario infatti **rappresentare i dati in modo tale da poter essere poi utilizzabili**. Inoltre, i dati potrebbero essere estratti e rappresentati in base al tipo di operazioni che devo fare su di essi. Bisogna inoltre **capire quale sia il formalismo da utilizzare** per rappresentare le informazioni. Se si hanno obiettivi diversi si hanno esigenze di estrarre le informazioni diverse. È necessario **capire quali siano le informazioni che bisogna estrarre dal codice**, ciò coincide con la prima operazione che si va a fare nel reverse engineering.

7.8.2 Models

Le **estrazioni identificate nei Goals sono analizzate per creare i modelli di rappresentazione**. Si individuano i **tipi di documenti che bisogna generare**, le **informazioni che devono essere derivate dal codice sorgente**, si **definiscono i modelli per rappresentare le informazioni e le loro relazioni** e si **producono infine i documenti desiderati**.

Sulla base dei requirements che si hanno nella fase di Goals, è **necessario prendere delle decisioni**. È quindi necessario ad esempio definire anche i modelli necessari per estrarre le informazioni e per costruire modelli.

È come se **goal sia la fase di requirements in cui si definiscono le necessità e le informazioni da estrarre**, poi è **necessario definire i modelli, come devono essere rappresentati e quali sono gli algoritmi che utilizzo** per estrarre le informazioni e costruire i modelli, ed eventualmente analizzarli. Potrei anche avere la necessità di costruire per strati. Si tira fuori

una rappresentazione e da essa si astrae un altro tipo di rappresentazione. Essendo automatizzato, documenti e design a più alto livello non possono essere estratti semplicemente andando a leggere un codice, ma **si richiede un intervento umano**.

Esempio: Un class diagram è basato sull'astrazione della classe, metodi ed attributi della classe, ovvero gli elementi che si vogliono estrarre. Il class diagram, inoltre, non dipende dal linguaggio del codice, dovendo però fare dei mapping differenti in base al tipo di linguaggio di programmazione usato. Ciò potrebbe essere utile nel caso in cui non si lavori con linguaggi project oriented. Sarà poi necessario capire la differenza tra classi dipendenti o classi associate. Una classe A è associata alla classe B se l'oggetto B che vado a chiamare è una variabile d'istanza della classe B e c'è un metodo della classe A che chiama un metodo di B. In base al tipo di linguaggio sarà necessario effettuare dei mapping differenti.

7.8.3 Tools

In questa fase i **tool necessari per fare reverse engineering sono identificati** e acquistati o sviluppati in-house. I tool sono principalmente suddivisi in due categorie:

- tools per l'**estrazione di informazioni e generare un program representation** in accordo con i modelli identificati
- tool per l'estrazione delle informazioni a la **produzione di documenti richiesti**. I documenti più ad alto livello non possono essere semplicemente estratti analizzando il codice, ma è **necessario l'intervento umano**. I tool per l'estrazione di informazioni lavorano generalmente sul codice sorgente o reconstruct design documents

7.9 Tecniche di facilitazione

Le tecniche usate per la facilitazione del reverse engineering sono:

- **lexical analysis:** Individua il **lessico del linguaggio**. Va ad **estrarre da un programma tutti i lexemi** che vengono usati, sulla base di quale sia il lessico del linguaggio. Il lessico del linguaggio sarà definito dalle **espressioni regolari**, mentre l'**automa regolare serve per riconoscerlo**. Si avrà quindi una **grammatica regolare**, tramite la quale si potrà esprimere il lessico del linguaggio.
- **syntax analysis:** in tal caso si ha **bisogno di un parser**. La syntax analysis **estrae i costrutti sintattici di un linguaggio che sono implementati all'interno di un programma**. La

sintassi di un linguaggio è espressa mediante **grammatica context free**. Per estrarre la rappresentazione sintattica di un programma uso un **parser che si basa su un automa regolare**. Lex prende la grammatica regolare ed automaticamente genera un automa regolare che mi consente di riconoscere se il lessico contenuto all'interno di un programma è corretto rispetto al lessico del linguaggio. Il parser userà un automa basato su stack. L'**albero di parsing può essere eseguito tramite tecnica top down o bottom up** in base al tipo di algoritmo. Una volta costruito l'**albero di parsing** possiamo trasformarlo in un **Abstract Syntax Tree (AST)**, potendo poi analizzarlo mediante control flow analysis.

- **control flow analysis:** ci sono due tipi di flussi di controllo:
 - **intraprocedurale:** mi riesce a tirare fuori **relazioni di flusso di controllo che esistono all'interno della singola funzione**, in modo da capire quale istruzione è eseguita dopo quale istruzione. Costruisco un **control flow graph** in modo tale da **individuare gli statement** a partire dal AST e le **relazioni del flusso di controllo tra gli statement**. Se c'è una struttura di controllo con una condizione ci saranno due archi uscenti, uno per la condizione vera una per la condizione falsa. Si può quindi navigare l'AST per tirare fuori delle informazioni
 - **interprocedurale:** vado a costruire un **call graph**. Si avrà un grafo in cui i **nodi saranno le funzioni e gli archi saranno le chiamate**. L'obiettivo è individuare gli **statement che legano le program unit**. Il call graph può essere **statico o dinamico**.
- **visualization:** il problema è visualizzare le informazioni estratte dal **call graph**. Per rendere efficace la visualizzazione delle informazioni è **necessario utilizzare algoritmi di layout**. Quest'ultimi però non bastano in quanto il call graph è molto intricato. Esistono perciò **tecniche chiamate bird eyes** che focalizzano su parti specifiche piuttosto che su altre. La visualization è quindi una strategia utilizzata per **permettere agli utenti di avere una maggiore comprensione del software system**. Due importanti tecniche di software visualization usando grafica 3D e virtual reality sono:
 - **representation:** **rappresentazione del singolo componente** mediante grafica o altri media
 - **visualization:** configurazione di un insieme di **rappresentazioni di informazioni individuali** non correlate che **costituiscono un higher level component**

- **program metrics:** quando utilizzo un tool di reverse engineering è possibile estrarre delle **metriche che potrebbero andare a misurare coesione, accoppiamento di una classe, profondità dell'albero di inheritance...** Si avrà quindi che l'estrazione di metriche userà le stesse tecniche che si usano per fare reverse engineering e tirare fuori rappresentazioni più alte. La **coesione di una classe è misurata mediante LCOM** nei linguaggi object oriented. La lack of cohesion in methods (LCOM) **crea un grafo bipartito** e tira fuori una misura che indica quanto i metodi della classe accedono agli attributi della classe. Se il grafo è completo, ovvero i metodi accedono agli attributi, la LCOM è molto bassa, mentre se ci sono meno archi, per cui alcuni metodi accedono ad alcune variabili e non a tutte, questo provoca la LCOM a crescere.

Esempio: la coesione di una classe è misurata mediante LCOM nei linguaggi object oriented. La lack of cohesion in methods (LCOM) crea un grafo bipartito e tira fuori una misura che indica quanto i metodi della classe accedono agli attributi della classe. Se il grafo è completo, ovvero i metodi accedono agli attributi, la LCOM è molto bassa, mentre se ci sono meno archi, per cui alcuni metodi accedono ad alcune variabili e non a tutte, questo provoca la LCOM a crescere. Per ogni attributo si calcola la percentuale di metodi che usano quell'attributo. Si va a vedere quanti sono i metodi che usano l'attributo rispetto a quanti sono i metodi. Si fa la somma e la si divide per il numero di attributi. Se per ogni metodo esso utilizza tutti gli attributi, allora la percentuale è 100% avendo valore pari ad 1. Se ciò accade per ogni attributo la somma di quest'ultimi sarà pari ad n (ovvero il numero di attributi), che fratto il numero di attributi darà 1. L'operazione successiva è: numero di archi - numero calcolato. Maggiore sarà il numero di archi più questo risultato tenderà a 0. Per cui la minore è la percentuale di metodi che usano un attributo più la LCOM si avvicinerà ad 1. Avendo classi con metodi statici, la coesione non può essere misurata su quanta collaborazione c'è tra i metodi. L'idea è che se dei metodi hanno a che fare con gli stessi concetti allora essi avranno gli stessi identificatori, per cui si possono usare informazioni semantiche. Si può, quindi, andare a calcolare la somiglianza testuale tra le rappresentazioni di tutti i metodi, facendo poi la media. Quando il valore sarà 1 vuol dire che si avrà una coesione bassa, mentre più il valore scende maggiore sarà la coesione. Un'altra tecnica sarebbe usare l'accoppiamento. Tutte queste emtriche però sono estraibili analizzando il codice. Le tecniche sono le stesse

7.10 Decompilation e Compilation

La decompilation è un qualcosa che serve a fare reverse engineering. Esso **permette di passare da un livello di astrazione più basso ad un livello di astrazione più alto**. Il processo di **compilazione e di produzione** di un eseguibile **non vengono considerati processi di forward engineering poichè automatizzati**, essendo assente un umano che dice cosa fare, mentre la decomilation inverso è parte del reverse engineering. Tale operazione potrebbe essere **fatta per recuperare codice sorgente o correggere errori**. Si potrebbe in alternativa avere codice binarie di file differenti, non sapendo come farli comunicare, per cui si effettua la decompilazione, estraendo il codice da entrambi e facendoli comunicare lavorando su codice sorgente. Tutte queste operazioni sono fatte in assenza di codice sorgente

7.11 Data reverse engineering

Si cerca di **ricostruire un modello dei dati da un sistema esistente**, eventualmente per fare reengineering. Ci possono essere diversi livelli:

- **livello fisico:** si parte dai programmi, parto dai file che contengono i dati e parto dal **data description language** che posso avere all'interno di un DBMS relazionale. Il risultato è lo schema logico
- **livello logico:** in questa fase lo schema concettuale è definito un modello semplice
- **livello concettuale:** in questa fase gli **user requiremetns sono forniti**, studiati e formalizzati in uno schema concettuale

7.11.1 Fasi del data reverse engineering

Le fasi nella data reverse engineering si dividono in:

- **data structure extraction:** estraiamo degli schemi logici, partendo dai dati che possono essere estratti da **schema fisico**. Alcune tecniche di estrazione sono:
 - **Analisi del testo DMS-DDL:** le istruzioni di dichiarazione della struttura dei dati in un dato DDL, che si trovano negli script dello schema e nei programmi applicativi, vengono analizzate per produrre uno schema logico approssimativo.
 - **Analisi del programma:** significa analizzare il codice sorgente al fine di rilevare vincoli di integrità e prove di strutture di dati aggiuntive.

- **Analisi dei dati:** ciò significa analizzare i file e i database per **identificare le strutture dei dati e le loro proprietà**, vale a dire campi univoci e dipendenze funzionali nei file e verificare ipotesi
 - **Integrazione dello schema:** all’analista vengono generalmente presentati diversi **schemi** durante l’elaborazione di più di una fonte di informazioni. Ciascuno di questi **schemi multipli offre una vista parziale degli oggetti dati**. Tutte queste viste parziali si riflettono sullo schema logico finale tramite un processo di integrazione dello schema.
- **data structure conceptualization:** estraiamo uno schema concettuale. Tale processo si suddivide in:
 - **make the schema ready:** lo schema originale potrebbe utilizzare alcuni **costrutti concreti**, come file e chiavi di accesso, che potrebbero essere stati utili nella fase di estrazione della struttura dei dati, ma ora **possono essere eliminati**. Alcuni **nomi possono essere tradotti in nomi più significativi** e alcune parti dello schema potrebbero essere ristrutturate prima di tentare di interpretarle.
 - **schema untraslation:** Ottengo un **misto tra schema logico e schema fisico**. Il risultato di tale fase è lo **schema concettuale grezzo** che devo andare a **normalizzare** per ottenere lo **schema concettuale normalizzato**. Per **tradurre da schema concettuale a schema logico si applica un mapping**, per cui è necessario eseguire un’operazione inversa.
 - **schema de-optimization:** io ho uno schema ottimizzato. Durante la fase di ottimizzazione potrei aver estratto delle relazioni fra entità per evitare join molto lunghi, ma questo non serve nello schema concettuale.

Optimization e normalization sono similari in quanto nella concettualizzazione si **rimpiazzano alcune entità da relazioni**, oppure **si riescono a costruire entità**, oppure **si rendono delle relazioni esplicite** o in alternativa **si vanno a standardizzare i nomi**. Si arriva infine ai tool per il reverse engineering che possono essere in diversi formati. Anche in questa fase non tutto può essere automatizzato

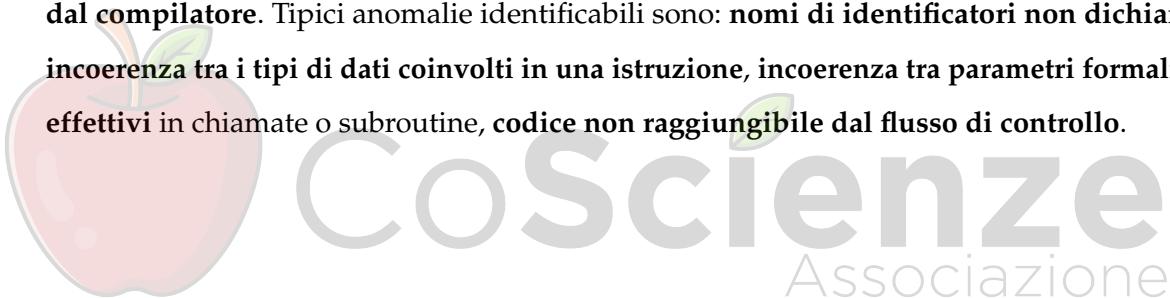
7.12 Analisi statica nella compilazione

I compilatori effettuano un’**analisi statica del codice** per verificare che un programma **soddisfi particolari caratteristiche** di correttezza statica, per poter generare il codice oggetto.

Alcuni tipi di analisi sono:

- **analisi lessicale:** consiste nell'identificazione dei **singoli elementi** (token) **componenti il programma** (keywords, identificatori, simboli del linguaggio)
- **analisi sintattica:** consiste nell'**esaminare le relazioni tra gli elementi identificati durante l'analisi lessicale**, che devono obbedire alle regole della grammatica del linguaggio
- **analisi semantica:** **rileva altri errori**, come l'utilizzo di variabili non dichiarate, effettua il controllo dei tipi nelle espressioni

I tipi di errori segnalati dal compilatore dipendono sia dalle **facility del linguaggio** che **dal tipo di compilatore in sè**. Avendo regole di visibilità statiche dei nomi, potrei permettere una maggiore rilevazione di anomalie. La generazione di **Cross Reference List** risulta molto utile in successive analisi del codice per l'**individuazione di anomalie non rilevabili dal compilatore**. Tipici anomalie identificabili sono: **nomi di identificatori non dichiarati**, **incoerenza tra i tipi di dati coinvolti in una istruzione**, **incoerenza tra parametri formali ed effettivi in chiamate o subroutine**, **codice non raggiungibile dal flusso di controllo**.



CAPITOLO 8

Grafi e modelli di rappresentazione dei programmi

8.1 Grafi

Un **grafo diretto** (o orientato) è una coppia (N, E) dove N è un insieme di nodi e E incluso o uguale a $N \times N$ è un insieme di archi.

8.2 Cammini e cicli

Per ogni arco (n, m) in E , n è detto un predecessore di m ed m è detto un successore di n . Un nodo può avere 0 o più successori e 0 più predecessori.

Un **cammino** di lunghezza $k-1$ è una sequenza di nodi $\langle n_1, n_2, \dots, n_k \rangle$ tali che per ogni i $1 \leq i \leq k$ (n_i, n_{i+1}) appartiene ad E . Un **ciclo** o **ciclo** è un cammino $\langle n_1, n_2, \dots, n_k \rangle$ tale che $n_k = n_1$.

8.3 Raggiungibilità di grafi non orientati

Un nodo m si dice **raggiungibile da un nodo n** se esiste un cammino di lunghezza $k-1$ tale che $n_1 = n$ ed $n_k = m$. Se esiste un cammino da n ad m vuol dire che m è raggiungibile a partire da n .

Un grafo **orientato** si dice **connesso** se ogni suo nodo è raggiungibile da ogni altro nodo sul grafo che si ottiene **eliminando l'orientamento degli archi**.

Un grafo orientato si dice **fortemente connesso** se ogni nodo è raggiungibile da ogni altro nodo (ossia se esiste un ciclo che coinvolge tutti i nodi del grafo).

8.4 Control flow graph

Il **grafo del flusso di controllo** di un programma P è una quadrupla $GFC(P) = \{ N, E, ni, nf \}$.

(N, E) è un **grafo diretto con archi etichettati**. L'insieme di nodi, togliendo n_i ed n_f (ovvero nodo iniziale e nodo finale) può essere partizionati in due sottoinsiemi:

- **Ns:** nodi che rappresentano **statement**
- **Np:** nodi che rappresentano **predicati** delle le strutture di controllo

$$ni \in N, nf \in N, N - \{ ni, nf \} = N_s \cup N_p$$

N_s e N_p sono **insiemi disgiunti di nodi** ossia $N_s \cap N_p = \emptyset$. Gli archi sono risultati del prodotto cartesiano:

$E \subseteq (N - nf) \times (N - ni) \times \{ true, false, uncond \}$ dove $\{ true, false, uncond \}$ rappresenta la label. Essa rappresenta la relazione di flusso di controllo.

Ciò significa che il **nodo finale non può avere un arco uscente** e un **nodo iniziale non può avere un arco entrante**. Ogni arco può essere poi etichettato con true, false ed uncond. **True e false sarà il valore degli archi che escono dai predicati**, mentre **uncond saranno i valori degli archi che escono dalle istruzioni**, questo perchè ogni statement ha un solo successore mentre un predicato ha due successori.

ni ed nf sono detti **nodo iniziale** e **nodo finale**. Un nodo in $N_s \cup ni$ ha un solo **successore immediato** ed il suo arco è etichettato con **uncond**. Un nodo in N_p ha due successori immediati e i suoi archi uscenti sono etichettati rispettivamente con **true** e **false**.

8.5 GFC ben formati

Un $GFC(P)$ è ben formato se esiste un cammino dal nodo iniziale ni ad ogni nodo in $N - \{ ni \}$ e da ogni nodo in $N - \{ nf \}$ al nodo finale nf . Un $CFG(P)$ è ben formato se ogni nodo sarà raggiungibile dal nodo iniziale ed ogni nodo può raggiungere il nodo finale. Un nodo che non sarà raggiungibile dal nodo iniziale verrà etichettato come **codice morto**. Un cammino o **cammino totale** sarà quindi un cammino da ni a nf

8.6 Costruzione del control flow graph

Il grafo è costituito secondo la seguente notazione:

- **nodo:** rappresenta un'**istruzione** identificata da un numero
- **arco:** rappresenta il **passaggio del flusso di controllo** etichettato con { vero, falso, uncond }

Il CFG può essere costruito combinando opportunamente insieme i grafi relativi alle strutture di controllo. Quando si **espande una struttura di controllo si riconosce il predicato e si riconoscono i vari percorsi**. La sequenza interna del predicato dovrà poi essere espansa. Nel caso in cui ci siano **go-to**, **break** e **continue**, non essendo strutturato il programma, **utilizzando un parser top down non è possibile costruire un CFG**. Bisognerà necessariamente usare un parser bottom up o in alternativa bisogna costruire un **AST**, ed analizzarlo. Nel caso in cui il **programma sia strutturato potrei usare anche un parser top-down per costruire il CFG**. Se la condizione è **delocalizzata in quanto sono presenti break, continue e go-to**, sarà **difficile costruire anche l'invariante di ciclo**.

Con il CFG **sequenze di nodi possono essere collassate in un solo nodo**, purché nel grafo semplificato **vengano mantenuti tutti i branch** (punti di decisione e biforcazione del flusso di controllo). Tale nodo può essere etichettato con i numeri dei nodi in esso ridotti. Vengono mantenuti **inalterati i cammini**, potendo quindi eseguire la stessa analisi, ma il CFG **risulterà più compatto**. Nel caso si voglia fare un **analisi di data flow non sarà possibile semplificare la struttura**. Il CFG può essere **esaminato per identificare ramificazioni del flusso di controllo e verificare l'esistenza di eventuali anomalie quali codice irraggiungibile, cattiva strutturazione o riconoscere che una struttura sia one in e one out**.

8.7 Analisi del flusso dati

Per tale analisi è **necessario aggiungere altre informazioni rispetto a quelle presenti nel CFG**. Per ogni statement è necessario sapere quali sono le variabili definite, a cui viene assegnato valore. Gli stati delle variabili sono:

- **definizione(d):** assegna il valore della variabile. La definizione spesso **ricade nell'istruzione di input o istruzioni di assegnamento**. A prescindere dal tipo di passaggio della variabile, essa se passata ad una funzione ricade in questa fase

- **uso(u)**: il valore della variabile è usata all'interno di un'espressione. Esistono vari tipi di usi:
 - **computazionale**: calcolo l'espressione per cui uso il valore dell'espressione perché il valore dell'espressione deve essere assegnato ad un'altra variabile o mandata in output
 - **predicativo**: uso che si fa di una variabile nel calcolare un'espressione condizionale che va a condizionare il flusso di controllo, variabile usata all'interno dei predicati delle istruzioni di controllo
- **annullamento(a)**: momento in cui la variabile non è più utilizzabile, per cui non può essere né usata né definita. Al termine del blocco il valore assegnato non è più significativo.

L'analisi del flusso dati va quindi a studiare sequenze di definizioni, usi e annullamenti sui cammini GFC. Sia la **definizione** che l'annullamento vanno a cancellare il valore che quella variabile aveva precedentemente. Nel caso di una nuova definizione di una variabile, la nuova definizione uccide la vecchia, poiché si va a sovrascrivere il valore. Va bene avere un terminamento della definizione nel caso in cui essa viene utilizzata come output, ma per il resto, per operazioni intermedie non va bene definire la variabile senza mai usarla.

8.8 Regole del CFG

Alcune regole da rispettare sono:

- **R1**: l'uso di una variabile x deve essere sempre preceduto in ogni sequenza da una definizione della stessa variabile, senza annullamenti intermedi. Un uso non preceduto da una definizione può corrispondere al potenziale uso di un valore non determinato.
- **R2**: una definizione di una variabile x deve essere seguita da un uso della variabile x , prima di un'altra definizione o di un annullamento di x . Ciò non vale solo nel caso in cui la variabile definita sia una variabile di output, per cui serve poi in uscita e verrà utilizzata all'interno del programma chiamante. Una definizione non seguita da un uso corrisponde all'assegnamento di un valore non utilizzato e quindi potenzialmente inutile. Una definizione che segue una definizione non è sempre un difetto, ma è comunque sintomo di una cattiva strutturazione

Avere un tool che permette di analizzare queste cose è un **algoritmo che ci permette di riscontrare eventuali errori**.

8.9 Espressioni cammino/variabile

L'espressione relativa ad un cammino p di un programma P per la variabile x è indicata con $P(p;x)$. A ciascun nodo del CFG è possibile associare l'insieme delle variabili definite in esso, quello delle variabili usate e quello delle variabili annullate. È quindi possibile scrivere l'espressione $P(p;x)$ facendo riferimento a tali insiemi.

8.10 Espressioni regolari

Se io voglio invece esprimere per ogni variabile ciò che succede su tutti i possibili cammini in esecuzione, posso utilizzare espressioni regolari. Un'espressione regolare è definita a partire da un alfabeto $A = \{ a, d, u, - \}$ e dalle seguenti regole ricorsive:

- è stringa nulla ed è **espressione regolare**
- ogni simbolo di A è un'espressione regolare
- se e_1 ed e_2 sono espressioni regolari, allora lo sono anche le espressioni che si formano da queste con l'uso degli operatori **sequenza** ($.$), **alternativa** ($+$) e **ciclo** ($*$), quindi $e_1.e_2$, e_1+e_2 e e_1^* sono **espressioni regolari**.
- Nient'altro è un'espressione regolare.

Il simbolo $[n \rightarrow]$ indica tutti i cammini uscenti dal nodo n , nodo n escluso mentre $[\rightarrow n]$ indica tutti i cammini entranti nel nodo n , nodo n escluso

8.11 Algoritmi definizione/uso

Costruire espressioni regolari per tutte le variabili è di per sé costoso. Per capire se ogni definizione di una variabile ha un uso corrispondente o un uso è raggiungibile da una definizione, esistono due algoritmi:

- **reaching definitions:** voglio vedere ogni definizione fin dove si propaga. Tale algoritmo esplora il grafo, lo visita su tutti i percorsi. Questo algoritmo **parte da un nodo e cerca di propagare la definizione di una variabile x** , volendo vedere quali usi

raggiunge senza essere uccisa da una nuova definizione di x. **Su ogni percorso mi porto avanti un token**, una coppia (**punto di definizione, variabile**) che permetterà **di creare una tripla** quando viene trovato il punto in cui la variabile è utilizzata. La tripla sarà formata da (**punto di definizione, variabile, punto di utilizzo**). Nel nodo quindi che presenterà un uso per x, si andrà ad indicare qual'è stata la definizione di x che è stata successivamente usata in quel nodo. **Quando si arriva in un nuovo nodo che presenterà un uso per x, si andrà ad aggiornare**, all'interno della coppia **il punto di definizione**, in quanto la nuova definizione uccide la precedente, ma non la tripla che resterà inalterata. Il costo di un algoritmo per la visita di un grafo sarà lineare al numero di archi (quadratico rispetto al numero di nodi).

- **reachable use:** invece di visitare il grafo forward, **lo visita backword**, partendo quindi dall'**uso della variabile** per vedere quali definizioni possono raggiungere quell'uso. Nel momento in cui si trova la definizione su un percorso, non si propaga più la variabile, in quanto sullo stesso percorso, se c'è un'altra definizione, essa sarà sicuramente dalla definizione che abbiamo trovato. **Parto propagando gli usi e mi fermo quando trovo la definizione che raggiunge quell'uso.**

Una definizione viene quindi portata avanti fin quando non si trova una nuova definizione mentre un uso viene portato all'indietro finché non si trova una definizione. Su uno stesso percorso non possono esserci altre definizioni che raggiungono l'uso da cui siamo partiti. Il compilatore applica uno di questi due algoritmi, non costruendo espressioni regolari.

8.12 Program slicing

Il **program slicing** è a cavallo tra analisi statica e testing. Essa è una **tecnica di decomposizione di un programma** che fornisce una risposta alla domanda "quali istruzioni influenzano il calcolo della variabile v all'istruzione p", dove le istruzioni p sono le istruzioni che contribuiscono al calcolo di quella variabile.

8.13 Slice

La slice S è un sottoinsieme eseguibile di P e che preserva il comportamento di P rispetto al criterio di slicing (p, V). Una slice deve essere:

- **executable:** poiché estratta da un programma a sua volta eseguibile

- **backward:** poichè il calcolo viene fatto partendo dal punto del criterio di slicing e tornando all'indietro
- **statica:** facendo analisi statica.

Una slice la si ottiene come un **sottoinsieme eseguibile delle istruzioni del programma di partenza**. Si ottiene a partire dal programma originario ma **rimuovendo degli statement**. Esso deve eseguibile e deve rispettare il comportamento del programma principale rispetto ad un criterio di slicing denominato come (p, V) . In questa definizione, **p è il punto del programma rispetto a quale si va ad osservare il comportamento** e **V è il sottoinsieme di variabili del programma rispetto cui si va ad osservare il comportamento**. Nel punto **p**, lo stato delle variabili **V** deve essere lo stesso sia nel programma che nella slice. Lo slice deve rispettare il comportamento del programma principale rispetto le variabili **V**. La variabile **p** andrà ad indicare l'istruzione **fin dove si arriverà con la slice**. Esistono più slicing dello stesso programma che rispettano il criterio di slice ma esiste una sola slice minima. Calcolare la slice minima è un problema non decidibile, per cui si utilizza un algoritmo di approssimazione per calcolare la slicing.

8.14 Traiettoria di stato

una traiettoria di stato di un programma P è una sequenza di coppie $T = (p_1, s_1), \dots, (p_k, s_k)$ dove per ogni i :

- p_i è un nodo del CFG di P
- p_1, p_2, \dots, p_k è un cammino sul CFG di P
- s_i è una funzione che mappa le variabili di P sui valori che assumono immediatamente prima dell'esecuzione di p_i , ovvero una **funzione che associa alle variabili del programma, i valori che assumono prima dell'esecuzione di p_i**

In altre parole una traiettoria di stato di un programma è **una traccia della sua esecuzione che evidenzia i valori delle variabili prima dell'esecuzione di ogni istruzione**

8.15 Criterio di slicing

Un criterio di slicing è una coppia $C = (p, V)$ dove p è un'istruzione di P e V è un insieme di variabili di P . Un criterio di slicing determina una funzione di proiezione Proj_C .

dove C è il criterio di slicing, che elimina da una traiettoria di stato tutte le coppie che non iniziano con p e restringe la funzione di mapping s alle sole variabili in V. La funzione di stato è quindi ristretta alle sole variabili

Sia T = $(p_1, s_1), \dots, (p_k, s_k)$ una traiettoria di stato di un programma P, allora

$\text{Proj}_{(p,V)}(p_i, s_i) = \text{vuoto}$ se p_i diverso da p

$\text{Proj}_{(p,V)}(p_i, s_i) = (p_i, s_i | V)$, ovvero (p_i, s_i limitato a V), se $p_i = p$

$\text{Proj}_{(p,V)}(T) = \text{Proj}_{(p,V)}(p_1, s_1), \dots, \text{Proj}_{(p,V)}(p_k, s_k)$

La proiezione viene applicata sia al programma che alla slice, per cui bisogna trovarsi la stessa cosa limitatamente alle variabili V. Noi eliminiamo tutte le coppie in cui p_i è diverso da p perché bisogna osservare cosa succede prima dell'istruzione p_i del criterio di slicing, ma esse vengono eliminate sia dal programma che dalla slice, dovendo dimostrare che lo stato sia lo stesso

8.16 Definizione formale di slice

Una slice di un programma P su un criterio di slicing C = (p, V) è **ogni programma eseguibile S con le due seguenti proprietà:**

- S è ottenuto da P mediante l'eliminazione di zero o più statement. Ciò indica anche che il programma è una slice di sé stesso
- Se P si arresta su un input I con traiettoria di stato T, allora anche S si arresta su input I con traiettoria di stato T' e $\text{Proj}_C(T) = \text{Proj}_{C'}(T')$, dove $C' = (\text{succ}(p), V)$ e $\text{succ}(p)$ è il più vicino successore di p che è in S, o p stesso se p è in S. Il nodo p potrebbe infatti non essere nella slice S che stiamo considerando, per cui si considera il successore più vicino. Il fatto che le due proiezioni devono essere uguali indica che lo stato delle variabili in V deve essere lo stesso, sia nel programma prima di p che nella slice prima del successore di p.

Possono esistere più slice per un programma P e un criterio di slicing C. Ogni programma ha almeno una slice rispetto ad ogni criterio di slicing C, in quanto si avrà sicuramente il programma stesso

8.17 Calcolo della slice minima

Il calcolo della slice minima è un problema di indecidibilità. Nella pratica, a partire dall'istruzione p, si vanno a calcolare tutte le istruzioni che influenzano direttamente o indi-

rettamente i valori che assumono le variabili in V immediatamente prima dell'esecuzione di p . Una tecnica alternativa è usare un algoritmo di approssimazione che va a calcolare la minima soluzione in un insieme di data flow, legate al control flow graph.

8.18 Slicing e Program Dependence Graph

Il calcolo di una slice si riduce ad un problema di raggiungibilità all'indietro (backward) sugli archi di un **program dependence graph**. Questo algoritmo iterativo va quindi a costruire il concetto di **reachable uses**. In pratica a partire dall'istruzione p va a calcolare tutte le istruzioni che influenzano direttamente o indirettamente i valori che assumono le variabili in V prima dell'esecuzione di p . Ci si basa su definizioni ed usi per andare a propagare all'indietro, ma non sono solo le funzioni uso ad influenzare il valore della variabile. Se si vuole calcolare la slice rispetto ad un'istruzione che si trova all'interno di un else di un if o di un ciclo, in generale ad una **struttura di controllo**, il valore della variabile dipende anche dal valore che assume il **predicato**, avendo quindi due tipi di dipendenze:

- dipendenza sul controllo
- dipendenza sui dati

Si va a considerare il valore di un'istruzione da cosa dipende, da quali istruzioni dipende. Non si dipende solo dalle istruzioni che vanno a calcolare in maniera transitiva il valore della variabile, ma anche dalle dipendenze sul controllo. Entrambe quindi vanno considerate in un algoritmo di slicing. Se tali dipendenze sono espresse sul CFG, si andranno ad aggiungere **data flow dependencies** se il nodo in cui viene usata la variabile dipende dal nodo in cui è definita, mentre si avranno anche casi in cui tutte le istruzioni che sono in un predicato di controllo dipendono dal controllo sul predicato dell'if. Se si riesce a costruire tale grafo, l'algoritmo di slicing diventa un algoritmo di raggiungibilità sul grafo. Per dire quali sono le istruzioni che influenzano il valore delle variabili usate in un certo punto, bisogna solo andare a ritroso e trovare tutti i nodi da cui quello che stiamo considerando dipende sul **data flow graph** e **control flow graph**.

Nell'algoritmo non è però definito da nessuna parte che l'insieme delle variabili che io considero nel criterio di slicing siano usate nell'istruzione che fa parte del criterio di slicing, mentre nell'algoritmo sopra specificato, si parte da un'istruzione per cui si parte dalle variabili che vengono utilizzate, o anche una sola variabile e non un insieme di variabili, basta che sia utilizzata in quell'istruzione.

8.19 Dipendenze sul controllo

Se ho un programma strutturato, calcolare il **data dependency graph** non è difficile, usando **reaching definition** o **reachable uses**, calcolandomi tutte le triple, ovvero tutti gli archi di **data flow** che sono all'interno del CFG. Il problema sono le dipendenze sul CFG. **Mediante programma strutturato, il nesting delle strutture di controllo mi dà anche le dipendenze sul controllo**, ma i programmi non sono strutturati.

In un **programma non strutturato**, si applica un algoritmo di **post dominanza**. Dato un CFG = (N, E, n_i, n_f) e due nodi n ed m in N:

- **n domina m in un grafo se partendo dal nodo iniziale n_i per arrivare ad m devo sempre passare per n**
- **m postdomina n se ogni cammino da n ad n_f contiene m**
- **m è dipendente sul controllo da n se:**
 - esiste un cammino (p_1, \dots, p_k) con $n = p_1$ e $m = p_k$ tale che m postdomina p_i per $1 < i < k$. Si sta quindi dicendo che **da n posso raggiungere m e tutti i nodi intermedi che si trovano tra n ed m sono postdominati da m**.
 - m non postdomina n

Ogni cammino da questi nodi per raggiungere n_f deve passare per m, ma m non postdomina n. Si è quindi **postdominati direttamente da un nodo, spesso un nodo predicato**. Informalmente questo significa che il nodo n rappresenta un predicato ossia ha due archi uscenti:

- seguendo un arco **m viene sicuramente eseguito**
- seguendo l'altro arco **m non viene necessariamente eseguito**

Tutte le istruzioni che ad esempio si trovano su un ramo if dipendono da un controllo sul predicato if. Si può parlare inoltre di **dipendenza diretta o indiretta**. Tutte le dipendenze, sia sui dati che sul controllo vengono quindi considerati in maniera diretta o indiretta per calcolare la slice.

Esempio: Supponendo di avere un'istruzione all'interno di un if else. Tutti i nodi che sono prima dell'istruzione all'interno del corpo dell'if, sono postdominati da m, ma il corpo dell'if non è postdominato m in quanto c'è un percorso alternativo, ovvero l'else, che non passa per l'istruzione m. L'istruzione sarà, però, dominata sul controllo. Poiché l'istruzione non

postdomina l'if. Poichè m non postdomina n, ovvero il predicato, non può essere dipendente sul controllo dai nodi che vengono prima di n.

8.20 Control Dependence Graph

Un control dependence graph di un programma P è una **quadrula CDG(N, E, n_i , n_f)** dove:

- $N \cup \{n_i, n_f\}$ è l'insieme dei nodi
- $N = N_s \cup N_p \cup R$ dove N_s e N_p sono definiti come in CFG, ovvero nodi statement e nodi predicati e **R è un insieme di nodi di regione**.
- $E \subseteq N_p \cup R \cup \{n_i\} \times N \cup \{n_f\}$ è l'insieme degli archi rappresenta la **relazione di dipendenza sul controllo**
 - i nodi regione riassumono le dipendenze sul controllo, ossia raggruppano i nodi con dipendenze comuni. Le regioni servono a raggruppare tutti i nodi che sono dipendenti sul controllo da un certo nodo
 - di conseguenza i nodi $n \in N_p$ hanno al più due archi uscenti labellati true e false e di solito diretti verso nodi regione

Nel caso il **programma sia strutturato sia avrà un control dependency three**, alternativamente si ha un control dependency graph.

8.21 Dipendenze sui dati

Esistono **diversi tipi di dipendenze sui dati definite in letteratura**, in particolare la dipendenza sul flusso dati è quella che utilizzata per il calcolo di una slice. Dati un control flow graph $CFG = (N, E, n_i, n_f)$ di un programma P, due nodi n ed m in N ed una variabile v di P:

- **DEF(n)** è l'insieme delle variabili definite al nodo n
- **USE(m)** è l'insieme delle variabili usate al nodo m
- **DEF USE(P)** è l'insieme delle triple definizione-uso (n, m, v) dove v è definita in n ed usata in m ed esiste un cammino da n ad m (n ed m esclusi) su cui v non è ridefinita (def-clear path).

Si parla di un **def-clear path**, rispetto la variabile v, da n ad m quando una variabile v non viene mai ridefinita su nessuno dei nodi compresi tra n ed m (n ed m esclusi). Se si ha un def-clear path in cui v viene definito in n e viene usato ad m, esso prende il nome def-use path, potendo quindi poi creare la tripla (n, m, v).

8.22 Program dependence graph

Se si aggiungono le triple def-use al control dependency graph abbiamo il program dependence graph a cui possiamo applicare l'algoritmo di slicing, il cui obiettivo è calcolare l'insieme di nodi del PDG che influenzano ciò che viene utilizzato in un certo nodo.

8.22.1 Algoritmo di slicing basato su PDG

Un algoritmo di slicing potrebbe essere un algoritmo in cui: Si parte da un insieme Slicelist vuota e la Worklist contiene il nodo p. Finchè la worklist non è vuota prendiamo il nodo n dalla worklist e la inseriamo nella slice. Per tutti i nodi m che non appartengono alla Slicelist ed n è control dependent o data dependent su m, si aggiunge m alla Worklist. Se un nodo sta nella slicelist vuol dire che è stato già considerato, se sta nella worklist vuol dire che è stato già raggiunto ma non ancora considerato perchè non lo abbiamo ancora aggiunto alla slicelist

8.23 Backward vs forward slicing

Esistono due tipi di slicing:

- **Backward:** quali istruzioni influenzano il valore di una variabile prima dell'esecuzione di una certa istruzione. Devo sapere, andando a ritroso, tutte le istruzioni che vanno ad influenzare una certa istruzione
- **Forward:** quali istruzioni sono impattate dal valore di una variabile in un dato punto all'interno di un programma. Se andassi a modificare il modo in cui viene calcolata una variabile, potrei capire quali istruzioni dipendono da tale modifica. Questo tipo di analisi è importante per l'impact analysis.

8.24 Decomposition slicing

La decomposition slicing non definisce un punto del programma rispetto cui calcolare lo slicing, ma indica solo la variabile. Per ottenere la decomposition slicing basta fare l'unione delle slice calcolate rispetto a tutti i punti del programma rispetto ad una certa variabile. Sono state dimostrate diverse proprietà sulle slice di decomposizione:

- esiste un ordinamento parziale tra le slice di decomposizione di un programma. Se considero tutte le slice di decomposizione per tutte le variabili all'interno di un programma, posso trovare che alcune slices di decomposizione sono contenute in altre, per cui la slice di decomposizione più grande è calcolata sulla base della slice di decomposizione più piccola.
- una slice di decomposizione ha una slice complemento rispetto al programma. Posso calcolare la differenza, rispetto al programma, di una slice decomposizione. Ogni slice ha una sua slice complemento per cui non vuol dire che il complemento di una slice è l'insieme vuoto. Non è detto che ci sia una variabile che domina e quindi contiene tutte le altre, per cui non è detto che ci sia una slice di decomposizione che contiene tutte le altre.

Ciò può essere usato per capire quali parti di un programma possono essere utilizzate per calcolare altre.

8.25 Slicing interprocedurale

Essa considera le chiamate tra procedure. Una slice interprocedurale contiene statement di diverse procedure che influenzano i valori assunti dalle variabili del criterio di slicing (p, V) prima dell'esecuzione di p .

Nel primo algoritmo proposto da Weiser, la slice era imprecisa, poichè il problema è che calcolando la slice, se sulla slice c'è una chiamata a funzione, questa istruzione è una chiamata ad un sottoprogramma. Bisognerebbe scendere nel sottoprogramma per vedere le istruzioni ed eventualmente le variabili di input che influenzano la variabile del criterio di slicing. L'algoritmo di Weiser, scendendo e poi risalendo, non ricorda dove fosse la chiamata, per cui salendo, ritorna in tutti i punti in cui potrebbe essere chiamata la funzione, ma non nel singolo punto da cui la chiamata era partita, non interessandosi che alcune delle chiamate potrebbero non influenzare la variabile. Non si prende quindi in considerazione il contesto

chiamante quando si scende nelle procedure chiamate. Altri autori, hanno proposto un algoritmo più preciso basato sul **system dependent graph** (SDG).

8.26 System dependence graph

L'SDG è un'estensione a livello interprocedurale del PDG. Esso è così strutturato:

- nella prima fase non si scende nelle procedure chiamate, attraversando i nodi della chiamata. Nel caso in cui però la funzione sia chiamata da un'altra, si sale alla funzione chiamante senza scendere

Esempio: se sto in B e B chiama A non vado in A, ma se B è chiamata da C vado in C

- nella seconda fase scende nelle funzioni ma non risale ai call sites. Vado a vedere tutti i punti di chiamata di altre funzioni scendendo solo e non salendo mai.

L'SDG permette questo tipo di operazioni poichè, per ogni chiamata, considera tutti i parametri di input e output, considerandolo due volte se esso è un parametro sia di input che di output. Vado ad assegnare ai parametri formali i valori dei parametri attuali. Si considerano inoltre delle ulteriori variabili intermedie x_{in} quando bisogna fare un passaggio da chiamante a chiamato e x_{out} quando bisogna fare il passaggio da chiamato a chiamante, questo per la variabile x. Si avrà una dipendenza in quanto $x_{in} = x$.

Posso non scendere all'interno delle funzioni chiamate poichè in SDG si vanno a definire delle dipendenze sulle funzioni. Si sa che il valore in uscita di una funzione dipende da una variabile, questo fino a risalire alle variabili in ingresso. Grazie a tali dipendenze, si può evitare di scendere, in quanto so da cosa dipende una determinata variabile. Non si ha bisogno di scendere o risalire perchè si avrà una dipendenza tra i valori di output ed i valori di input dei parametri attuali

Per calcolare però le dipendenze, è necessario usare un algoritmo. Fondamentalmente si potrebbe usare slicing. Si sta cercando di capire, all'interno di una funzione, che relazioni di dipendenza ci sono tra i parametri formali, potendo riportare tali dipendenze sui parametri attuali. L'analisi delle dipendenze è preliminare, e una volta calcolate le dipendenze è possibile calcolare la slice.

8.27 Usi dello slicing

Lo slicing è stato utilizzato per il **riuso**, per l'**estrazione di funzioni**. Immaginando di avere un programma molto grande, un possibile smell è il **long method**. Una delle operazioni di refactoring è l'**extract method refactoring**, applicabile mediante slicing. Si usa quindi lo **slicing per fare downsizing di programmi procedurali molto grandi**. Si potrebbe dividere il programma in sottoprogrammi ed eseguire quest'ultimi in parallelo. **Sulla base delle slice che vado ad estrarre e sulla base delle dipendenze, si può capire quanto è parallelizzabile un programma.**

8.28 Client-Server restructuring

Possiamo inoltre effettuare una ristrutturazione client-server. Nel **client risiede l'interfaccia utente** e controlla l'esecuzione, mentre il **server implementa funzioni di business e di gestione del DB**. Tale operazione di slicing è divisa in due passi:

- **primo passo (automatico)**: individuazione della componente client
- **secondo passo (semi-automatico)**: ristrutturazione client/server

La tecnica di transform analysis parte dal centro del DFG e **crea una structure chart che normalmente è fatta a forma di albero**, dove però i **nodi interni sono nodi di coordinamento**. La structure chart permette inoltre di indicare se ci sono delle **major decisions** e dei **major loops**, ovvero se una funzione viene chiamata all'interno di un modulo di coordinamento ripetutamente.

8.29 Individuazione del criterio di slicing

Per capire il criterio di slicing, ovvero il **punto del programma in cui mettermi**, è necessario considerare se voglio una funzionalità interna o esterna:

- Volendo una **funzionalità esterna**, è necessario fermare l'algoritmo quando si raggiungono **istruzioni e variabili di input**. Se la coesione è scarsa nella funzione ho un'alta probabilità che le variabili della funzione saranno calcolate con istruzioni differenti. **Più la coesione è alta più le variabili dipenderanno l'una dall'altra**. Si continua a cercare fin quando non si trovano le **variabili di input da cui le variabili di output dipendono**.

- Volendo una **funzionalità interna** diventa più difficile. Per la funzione interna è **necessario mappare la specifica della funzione sul pezzo di codice**. La specifica di una funzione però è fornita mediante **precondizioni e postcondizioni**. Si andrebbe a **fare esecuzione simbolica e vedere quando la path condition che ho implica la postcondizione**. Se io trovo che la path condition implica la postcondizione, allora vuol dire che in quel momento ho calcolato la variabile. Se la path condition è vera, la post-condition, allora ho calcolato la variabile in base alla post-condition. Se io trovo che la path condition, ovvero la condizione che mi dice che io ho attraversato il path fino a quel punto, implica la post condizione, allora la post-condizione è vera, per cui ho attraversato le istruzioni che mi permettono di calcolare le variabili di output della funzione, per cui sarà lì che dovrò inserire il criterio di slicing. **Si torna a ritroso e ci si ferma quando si verifica la precondizione, avendo trovato la slice**. Per le internal functionalities mi aspetto che la parte di CFG che calcola tale funzione sia one in/one out. Il **criterio di slicing dovrebbe contenere in questo caso sia l'istruzione da cui parto fino all'istruzione in cui devo fermarmi**. Ciò sarà fatto usando la specifica. Posso a questo punto calcolare la slice

8.30 Slicing dinamico

Il debugging è stato il primo motivo per cui è stato definito il backward slicing, mentre l'impact analysis è il primo motivo per cui è stato definito il forward slicing. Avendo una failure, lo abbiamo su un caso di test. Calcolando la **slice statica**, andiamo ad estrarre anche **una serie di informazioni** che non centrano con la traccia di esecuzione che ha dato il caso di test. Con lo slicing statico abbiamo una serie di informazioni aggiuntive, **inutili al test**.

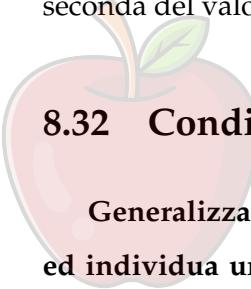
Per calcolare la slice più accuratamente è necessario quindi usare lo slicing dinamico. La slice dinamica preserva il comportamento del programma rispetto ad un particolare input e quindi rispetto ad un'unica traiettoria di stato. Si considera solo una singola traiettoria di stato, riuscendo a calcolare slice più piccole e ad essere più efficace nel caso di debugging.

Per applicare il PDG allo slicing dinamico, il modo più semplice è quello di **prendere il PDG statico e marcare le istruzioni che vengono eseguite e calcolare a ritroso le dipendenze**. Il problema è che si potrebbero avere **due nodi che vengono eseguiti**, ma la dipendenza che c'è tra questi due nodi **non fa parte del percorso di quello specifico caso di test**. Per essere specifici bisognerebbe **calcolare un PDG dinamico**, ovvero quello **calcolato sulla base della**

traiettoria di stato. Si calcola quindi prima la traccia di esecuzione e poi il PDG e calcolo le dipendenze.

8.31 Quasi static slicing

Se voglio calcolare una slice fissando il valore di certe variabili in input ma non il valore di altre? Facendo static slicing considero tutte le possibili traiettorie di stato, facendo dinamic solo una mentre con il quasi slicing si considera un sottoinsieme delle traiettorie di stato, in quanto alcune variabili non hanno un valore definito. Se con il dinamic slicing eseguo il programma per avere la traiettoria di stato, con quasi static slicing posso effettuare partial computation e partial evaluation, in cui alcuni valori sono fissati mentre altri sono simbolici, come un'esecuzione simbolica mista. A causa del valore definito di alcune variabili, il valore di verità di alcuni predicati può essere sempre vero o sempre falso. Se ho una variabile simbolica all'interno dell'espressione, il predicato potrebbe essere vero o falso a seconda del valore effettivo che quella variabile assume



8.32 Conditioned slicing

Generalizzazione di tutte le tecniche di slicing. Essa effettua un quasi static slicing ed individua un sottoinsieme delle traiettorie di stato, ma non è possibile individuare qualunque sottoinsieme traiettorie di stato rispetto cui fare slicing, ma soltanto quelle in cui alcune variabili hanno un valore di stato ed altre no. Con quasi static slicing posso circoscrivere la regione del mio input con una precondizione. Si fissa una condizione sulle variabili in input io circoscrivo una parte del mio spazio di input. La condizione può essere espressa in tutti i modi possibili.

L'idea è di esprimere condizioni più complesse che circoscrivono lo spazio di input. Con il simultaneous dynamic slicing, piuttosto che prendere un singolo caso di test, si prendono un insieme di casi di test, dimostrando che la simultaneous dynamic slice non è l'unione delle singole dynamic slice, ma ha fornito un algoritmo per calcolare simultaneamente la slice che rispettasse tutte le condizioni di input

La conditioned slicing usa any set of input. Tramite conditioned slicing posso calcolare sia simultaneous dynamic slicing che quasi static slicing. In generale, quando ho una condizione, il calcolo della conditioned slicing usa symbolic execution. Produce un conditioned program dependency graph in cui si taglia dal programma tutto ciò che non rispetta la

condizione sulle variabili di input. Calcola il PDG che io ottengo dal conditioned program e poi produco la conditioned slicing facendo static slicing sul conditioned program dependency graph

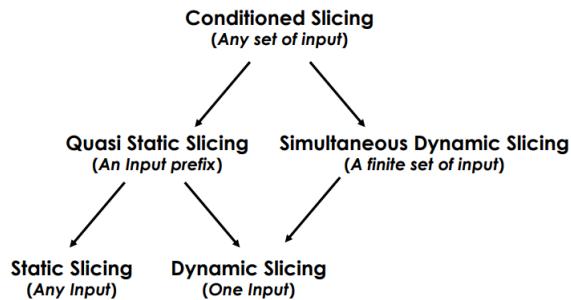


Figura 8.1: Conditioned slicing

8.33 Amorphous program slicing

Di solito lo slicing si ottiene eliminando gli statement. Con questo tipo di slicing si applica una trasformazione diversa, come semplificazione di codice, che preserva il comportamento.

8.34 Problema dello slicing

Il problema dello slicing è legato a problemi di indecidibilità, per cui ci sono pochi tool. Quelli che funzionano meglio sono i **tool di slicing dinamico**, poichè, lavorando su una traccia in esecuzione, permette di lavorare in modo più facile. L'**analisi dinamica è più facile dell'analisi statica e richiede maggiore infrastruttura rispetto ad analisi statica**. Con l'analisi dinamica si applicano le stesse tecniche di analisi statica, ma sulla traccia di esecuzione. Per fare ciò è necessario **strumentare il codice**

CAPITOLO 9

Impact analysis

9.1 Introduzione: Impact analysis

Con l'impact analysis si va a determinare se ci sono parti critiche, se ci sono parti rischiose da impattare, per determinare la storia dei cambiamenti, per la valutazione di futuri cambiamenti, per comprendere e per determinare come gli elementi che verranno cambiati sono legati e per determinare le porzioni del software che devono essere sottoposte a regression testing.

Se voglio risparmiare sui costi del testing posso andare a vedere quali sono i casi di test che devo rieseguire, ovvero parti che vanno ad esercitare componenti impattate da una modifica. Parlando di impact analysis non andiamo solo a vedere il componenti di codice impattate, ma anche gli artefatti di più alto livello, per cui è importante il concetto di tracciabilità. La tracciabilità può afferire a qualsiasi artefatto e può essere di due tipi:

- **horizontal traceability:** detta anche **external**, significa che è una **tracciabilità tra artefatti a diversi livelli di astrazione**, come tra requisiti e design, tra requisiti e casi di test...
- **vertical traceability:** detta anche **interna**, all'interno dello stesso livello di astrazione, tra componenti di codice, tra requisiti...

Uno degli aspetti dell'impact analysis è l'**impact effect analysis**, ovvero l'**analisi degli effetti collaterali**, ovvero obiettivo principale dell'impact analysis. In quest'ultima si vogliono

andare ad effettuare due operazioni: cercare le componenti che devo modificare per implementare la funzionalità richiesta e individuare quali sono i componenti che devo andare a modificare come conseguenza della modifica che vado a fare. Alcuni componenti non si vanno a modificare perchè essi devono implementare la change request, ma poichè a causa di altri elementi per implementare la CR devono essere modificati a loro volta. Possiamo misurare quanti ripple effect ci sono come conseguenza di una modifica, riuscendo a capire quanto sta aumentando la complessità del software.

9.2 Processo di impact analysis

Per prima cosa si prende una CR e si effettua un'analisi della change request. In tal modo vado ad:

- individuare quali sono le componenti impattate direttamente dalla change request, che devono essere modificate per implementare la change request. L'insieme di componenti viene definito Starting Impact Set (SIS)
- Si va a vedere, mediante ripple analysis, se oltre agli elementi da modificare per implementare la CR, c'è altro da modificare indirettamente, parlando dell'Candidate Impact Set (CIS). Esso è il SIS arricchito dai componenti che devo andare a modificare come effetto della ripple analysis
- Si vanno ad implementare poi i cambiamenti richiesti. Potrebbe succedere che alcuni dei componenti che avevo previsto si modificassero, non sono da modificare. Tali componenti che non devono essere modificati vengono messi nel False Positive Impact Set (FPIS), mentre i componenti che durante la modifica sono stati scoperti come da modificare, andranno nel Discovered Impact Set (DIS). L'insieme delle componenti modificate mi dà l'Actual Impact Set (AIS). Sottraendo da CIS l'insieme AIS posso ottenere il DIS

Il processo potrebbe essere iterativo, potendo tracciare degli impatti nella fase finale o continuare le implementazioni fin quando la richiesta non è rispettata.

9.3 Tipi di tracciabilità

La tracciabilità può essere implicita o esplicita. Esplicita se si ha la matrice di tracciabilità che viene aggiornata di volta in volta, mentre implicita se si hanno ad esempio delle

dipendenze tra componenti di codice che posso individuare tramite parsing. In tal caso non conviene mantenere la matrice di tracciabilità ma calcolare la dipendenza ogni volta.

Ci possono essere diversi modi di calcolare la tracciabilità, come **syntax based** (analisi del codice), **knowledge based** (information retrieval). Si potrebbero avere anche **dipendenze logiche tra componenti**, non ottenibili tramite parsing. Sarebbe quindi necessario **avere informazioni storiche delle versioni** se, ad esempio, due componenti ogni volta che viene modificata una viene modificata anche l'altra. Dire che il **codice è tracciabile** vuol dire quindi **che il codice ha identificatori, commenti...** che sono **maggiormente riconducibili ai requisiti**, per cui la somiglianza testuale è più alta.

Esempio: Per calcolare la tracciabilità tra requisiti e codice, potrei usare una matrice o calcolarla di volta in volta, calcolandola mediante tecniche di information retrieval. Userei i requisiti come query e andrei a vedere quali parte del codice somiglia di più al requisito. A differenza del codice, l'output dell'information retrieval potrebbe contenere garbage o falsi positivi, per cui questo processo è sempre automatizzato.

9.4 Metriche dell'information retrieval

Per misurare la qualità dell'impact analysis, si è proposto di usare come metriche la precision e recall:

- **Recall:** frazione di documenti rilevanti che sono stati tirati fuori dalla query.
- **Precision:** mi serve per capire quanti pochi documenti non interessanti ho nel mio **result set**. La precision rappresenta la frazione di documenti estratti che è effettivamente rilevante. Tale valore andrà da 0 ad 1. Pur avendo **precisione pari ad 1 bisogna stare attenti**, in quanto i documenti da cui si partiva potrebbero essere pochi.

Normalmente c'è un trade off tra precision e recall. **Più alta è la precision minore è la recall e viceversa.** Nell'ambito dell'impact analysis si ragiona come:

- **Recall:** frazione di impatti effettivi contenuti nel CIS, ovvero l'intersezione tra CIS e AIS fratto AIS
- **Precision:** rappresenta la frazione di impatti candidati che sono effettivamente impattati quindi l'intersezione tra CIS e AIS fratto CIS

Queste metriche vengono quindi **effettuate a valle della modifica**. Le metriche successive **si basano sul concetto che se l'impact analysis non tira fuori tutto l'AIS non è buono**, per

cui si può ragionare in termini di precisione solo se la recall è pari ad 1. Le metriche che si basano su tale concetto sono **adequacy** ed **effectiveness**

9.5 Adequacy

Abilità dell'approccio di **individuare tutti gli impatti attuali, ovvero AIS è contenuto o uguale a CIS**. La metrica che si usa per l'adequacy è l'**inclusiveness**, per cui **esso può assumere valore 1 AIS è contenuto o uguale a CIS 0 altrimenti**. Pur se un singolo componente non è nel CIS allora l'**inclusiveness è 0**, ma può capitare spesso di dimenticare qualche componente, per cui il calcolo con tale metrica è pesante. **Se l'adequacy è pari ad 1**, ovvero se l'approccio è adeguato, **posso valutarne l'efficacia**.

9.6 Effectiveness

L'efficacia si misura in diverse caratteristiche:

- **Ripple sensitiviy:** va a valutare l'impatto del ripple effect. In pratica considera l'insieme di oggetti impattati dalla change request, chiamandoli **Direct Impacted Set of Objects (DISO)**, ovvero il **SIS**. Si va successivamente a vedere quali sono i componenti impattati direttamente a causa del ripple effect, tracciando gli **impatti candidati**, ovvero CIS che viene chiamato **Indirectly Impacted Set of Objects (IISO)**. La cardinalità di IISO è un indicatore del ripple effect. Più IISO è grande rispetto a DISO maggiore è il ripple effect. La metrica usata per calcolare ciò viene chiamata **amplification = IISO fratto DISO**. Vorrei che si **mantenesse un rapporto da 0 ed 1** per indicare che per ogni componente modificato direttamente si modifica al più un componente indirettamente
- **Sharpness:** Essa viene calcolata con il **changeRate**. Essa indica l'**abilità dell'impact analysis di evitare di includere oggetti nel CIS che non sono necessari per essere cambiati**. La **changeRate = CIS fratto Sistema**. Essa non mi dice quanto è buona l'impact analysis in realtà ma mi indica **quanto sta diventando complesso il sistema**. Più il **changeRate è elevato, maggiore è il numero di componenti che dovranno essere modificate**. Ciò però non afferisce con l'impact analysis, ma è un **problema del sistema**. Di per sé non mi dà una bontà della misura dell'impact analysis.

- **Adherence:** Si misura con **S-Ratio = AIS fratto CIS** ed indica l'abilità dell'approccio di produrre un CIS più vicino possibile ad AIS. La S-Ratio è esattamente la precision nel caso in cui AIS è contenuto nel CIS. Tale metrica viene applicata quando l'inclusiveness è pari ad 1

Inclusiveness ed adherence possono essere calcolati solo dopo aver effettuato la modifica, mentre il change rate e l'amplification possono essere calcolate prima

9.7 Identificare il SIS

Per identificare il SIS, **avendo solo il codice e non partendo dai requisiti**, è necessario **effettuare una concept assignment**. La Change request esprimrà dei concetti, per cui sarà necessario partire dai concetti della CR in linguaggio naturale e mapparli sul codice. Automatizzare completamente un'operazione simile è complesso. L'idea è quella di **cercare identificatori all'interno del codice**. Un esempio potrebbe essere il pattern "grep" che indica tutte le linee di codice che contengono uno specifico identificatore. Quello che si può usare in alternativa è **information retrieval**. **Si usa la CR come query e il codice come documento**.

Altre tecniche si ha nel caso si abbia una **test suite documentata**, in cui si ha anche un **mapping tra test cases e features** che questi test cases vanno ad implementare. Dal momento in cui la CR indica quali sono le features che voglio modificare, mediante relazione tra features e test cases, **individuo i test case che dovrebbero eseguire il codice che implementa le features**. Se io divido la test suite in test cases che testano le features che hanno a che fare con la CR e test cases che non hanno a che fare con features che interessano la CR, eseguo entrambe le volte il codice, **posso individuare gli statement eseguiti con i test cases che implementano le features e quelli che non implementano le features**.

Tecnica alternativa è quella **incrementale di partire dal main**, vedere se si ha una chiamata a funzione e vedere se quella funzione è impattata. Se non è impattata non si fa nulla e **se è impattata si entra nella funzione e si vede a sua volta se questa funzione chiama altre che potrebbero essere impattate**.

9.8 Analisi del traceability graph

Alternativa è quella di usare la **tracciabilità verticale o orizzontale**. Arrivando la CR, essendo simile ai requisiti, **si partirà dai requisiti**, in quanto applicare **information retrieval** tra CR e requisiti dovrebbe essere più facile. Si useranno i requisiti come base documentale

e la CR come query. Si applica la tracciabilità orizzontale, effettuando successivamente **ripple effect analysis**.

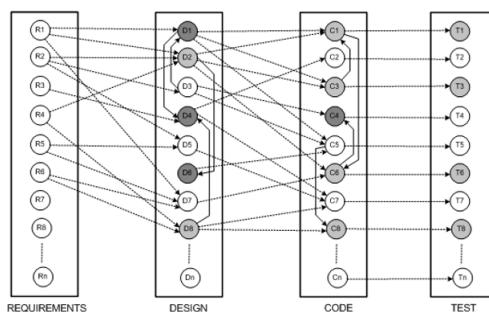


Figura 9.1: Analisi del traceability graph

9.9 Identificare il CIS

La tracciabilità **la uso per andare a trovare il CIS**. Gli impatti fra due componenti possono essere di due tipi:

- **Diretto:** se c'è un link di tracciabilità tra due componenti si ha un impatto diretto
- **indiretto:** se da un componente raggiungo mediante diversi link di tracciabilità un altro componente si ha un impatto indiretto

Per sapere se una componente ha un impatto su un'altra è necessario calcolare la **connectivity matrix**, la matrice di adiacenza di quel grafo. Da essa posso calcolare il grafo e la matrice di raggiungibilità. Si calcolano relazioni dirette, relazioni in due passi, in tre passi... facendo moltiplicazioni tra matrici.

9.10 Risoluzione della changeRate

Se la matrice di raggiungibilità è troppo folta, vuol dire che ho come CIS quasi tutto il sistema, ovvero avrò **changeRate** troppo elevato, per cui c'è qualcosa che non va. Per risolvere il problema posso usare:

- **distance based approach:** l'impatto vado a considerarlo fino ad un determinato valore, andando ad indicare nella matrice di raggiungibilità in quanti archi riesco a raggiungere un nodo, scartando quelli superiori alla soglia

- **incremental approach:** analizzo uno alla volta, come nel ripple effect, **usando delle euristiche o dei criteri per definire se in alcuni casi c'è un impatto o no.** Esso è meno automatizzabile rispetto al distance based.

9.11 Call graph

In un **call graph** un **nodo presenta una funzione**, un componente o un metodo mentre un **arco rappresenta una relazione in cui A invoca B**. Se viene effettuata **una modifica su un componente**, si va a propagare l'impatto sia **avanti che dietro**, ovvero sui nodi che chiamano quel componente e quelli chiamati dal componente. Se si ragiona così però si avrebbe un **impact set impreciso**. Si dovrebbe quindi considerare **anche la parte dinamica**, ovvero quando chiamo un nodo, chi effettivamente viene chiamato? Si va quindi a **considerare una traccia di esecuzione**. Mediante tale traccia **possiamo individuare quali sono le procedure che sono indirettamente o direttamente invocate**, ma anche le procedure che sono invocate dopo che il componente termina.

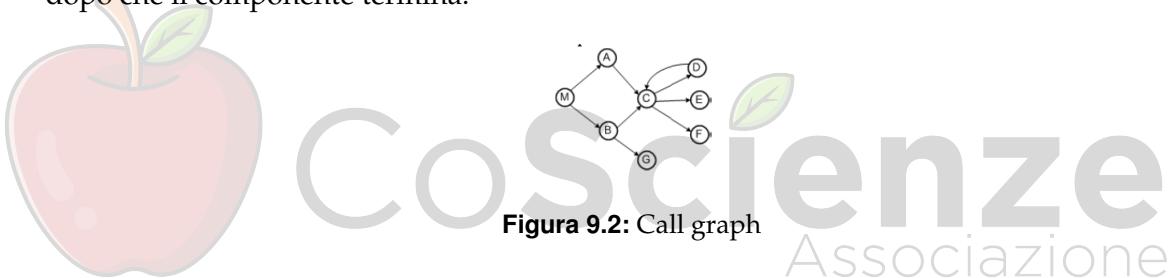


Figura 9.2: Call graph

9.12 Program dependence graph

Nel program dependency graph:

- Ogni statement è rappresentato da un nodo
- ogni predicato è espresso da un nodo

Abbiamo un qualcosa di più a **grana fine**. Si hanno le dipendenze. In tal caso il PDG non ha nodi regione e le **dipendenze sono al contrario**. Se si vuole calcolare il **backward slicing** le **dipendenze vanno seguite secondo il loro verso e non al contrario**. Tecniche che possono essere usate sono:

- **static program slice:** tecnica più imprecisa. Per una variabile var al nodo n, è possibile identificare tutte le definizioni che raggiungono var. Trova tutti i nodi nel PDG che sono

raggiungibili da quei nodi. I nodi visitati nel processo di attraversamento costituiscono la fetta desiderata.

- **dynamic program slice:** più efficiente nello localizzare errori

9.13 Ripple effect

Per calcolare il ripple effect all'interno del codice, modificando un'istruzione, ovvero dove la modifica ha impatto, **si potrebbe usare forward slicing o la tecnica della matrice**.

Con la **tecnica della matrice**, supponendo di avere un modulo m1 che viene chiamato da m2, per cui il valore di d in m1 influenza il valore di x in m2. **Supponendo si cambi il calcolo precedente, si andrà a vedere dove la modifica ha impatto.** Ciò viene espresso con delle matrici. Come prima cosa si va a **vedere se la variabile modificata è globale**. Se è globale allora si avrà un impatto differente. Se invece è una **variabile di input**, allora la propagazione sta dal **chiamante al chiamato**, nel caso in cui sia di **output** invece, la propagazione va dal **chiamato al chiamante**.

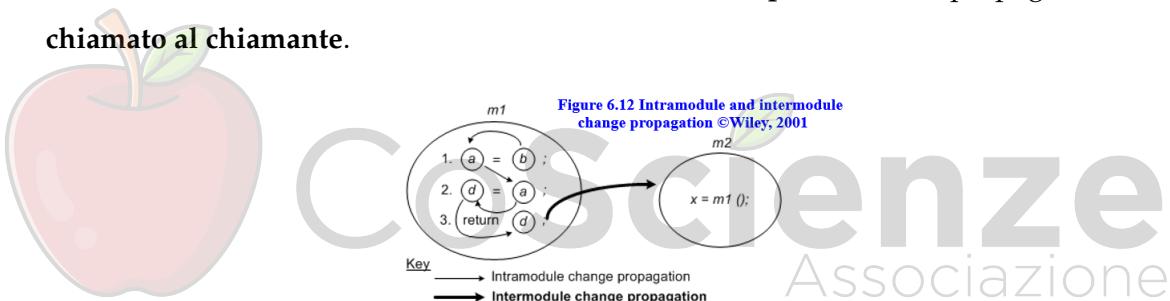


Figura 9.3: Call graph

CAPITOLO 10

Testing

10.1 Verifica e convalida



Il testing è una tecnica che serve a valutare la qualità di un sistema software. La verifica include test, ispezione ed analisi statica, mentre la convalida include test di usabilità, feedback dell'utente. La convalida si fa rispetto ai requisiti reali, ovvero quelli che non sono ancora specificati. La verifica richiede invece una sorta di confronto tra un output della fase di processo ed un altro output. Quando si fa un testing di sistema si sta verificando il sistema rispetto ai requisiti funzionali e non funzionali specificati. Presi i casi di test, si vede il comportamento del sistema, estraendo l'oracolo dai requisiti specificati. È la specifica dei requisiti che mi consente di derivare gli oracoli. La verifica risponde quindi alla domanda "stiamo costruendo il prodotto nel giusto". Se i requisiti però non sono stati specificati, anche se il sistema è verificato rispetto ai requisiti, il sistema non potrà essere convalidato. Alcune caratteristiche sono validabili, altre convalidabili, altre entrambe, ciò dipende da come sono state specificate.

L'usabilità è solo convalidabile poiché meno oggettiva rispetto alle altre caratteristiche di qualità, mentre l'efficienza dovrebbe essere verificabile, in base a come sono stati specificati i requisiti. Se un requisito definisce che il sistema deve rispondere in un determinato lasso di tempo allora l'efficienza è validabile. Posso verificare alcuni aspetti dell'usabilità con dei test, ma l'usabilità è convalidabile in quanto dipende da chi usa il sistema. Se i requisiti sono espressi in maniera vaga allora essi non sono verificabili. Per poter verificare devo poter

misurare. Se non esprimo in maniera chiara i requisiti non funzionali, in modo tale da poter estrarre delle misure e confrontarli con quello indicato nel requisito, allora il requisito non è verificabile.

Esempio: Se espressa male e non misurabile allora l'efficienza non è verificabile ma convalidabile

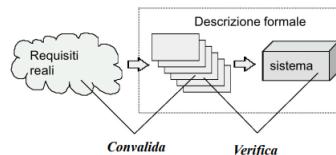


Figura 10.1: Verifica e convalida

10.2 Dependability

Essa contiene diverse caratteristiche di qualità come **correctness**, **reliability**, **safety** e **robustness**.

- molto più forte in sistemi safety-critical
- per i prodotti per il mercato di massa, la **dependability** è meno importante del **time to market**
- possono variare anche nella stessa classe di prodotti.

10.3 Verifica statica e dinamica

La verifica può essere sia statica che dinamica:

- **verifica statica (software inspection):** si effettua tramite **processi di review** durante il quale, nel caso del codice o nel caso di modelli formali, **possono essere applicate tecniche di analisi automatica**, per verificare staticamente certe proprietà. L'analisi statica è un **processo di valutazione di un sistema o di un suo componente basato sulla forma, struttura, contenuto o documentazione**
- **verifica dinamica (software testing):** si basa su un processo del sistema software o di un suo componente **basandosi sull'osservazione della sua esecuzione**. È necessario selezionare **dati di test per eseguire il sistema**. L'analisi dinamica è il **processo di valutazione di un sistema software o di un suo componente basato sull'osservazione**

del suo comportamento in esecuzione. Si ha bisogno sia di selezionare **casi di test** che di effettuare eventualmente delle strumentazione del codice. Si potrebbero ad esempio inserire dei probe, che permettono di analizzare lo stato del sistema in alcuni punti. Con la **verifica dinamica è necessario un oracolo**, in quanto si fa l'analisi della traccia di esecuzione in output, memorizzata in un file e lo si confronta eventualmente con il codice per dare delle indicazioni

La **verifica statica** può essere fatta su tutto mentre per la **verifica dinamica** si necessita di un qualcosa che deve essere eseguito.

10.4 Testing and debugging

Verification e validation riguardano la ricerca di difetti nel software. Si va a verificare la presenza di difetti all'interno del software. per **trovare dove sono localizzati ed eliminarli** si ha però **bisogno di un processo distinto**. Testing e debugging sono due processi distinti:

- **debugging:** riguarda l'individuazione e l'eliminazione dei difetti. Il debugging implica formulare ipotesi riguardo il comportamento del programma e quindi verificare queste ipotesi per localizzare gli errori.
- **testing:** è il processo di esecuzione del software con l'obiettivo di trovare malfunzionamenti. Un **test che non rileva malfunzionamenti** è un **test fallito**, mentre il testing non può dimostrare l'assenza di difetti ma può solo **dimostrare la presenza di essi** (tesi di Dijkstra). Il testing è il processo di analisi che un programmatore fa per **individuare dei bug nel proprio programma**. L'idea iniziale di **testing era quello tipo di esaustivo**, idea che con il passare del tempo è stata abbandonata. Il **testing non è una fase del processo di sviluppo, ma è un processo le cui attività si estendono per tutto il processo di sviluppo software** ed anche dopo, durante la manutenzione

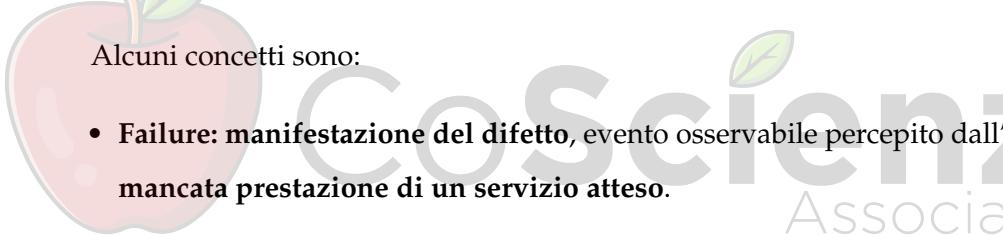
Si fanno le ipotesi e si testano le ipotesi. Le ipotesi afferiscono alla parte di codice in cui si trova un difetto, per cui si potrebbero mettere dei punti di ispezione prima e dopo la parte di codice. Si individua o localizza l'errore usando **test data**, ispezione e poi si progetta una tattica per correggere l'errore. Con il **testing si verifica se il software funziona o meno** mentre con il **debugging si vanno a localizzare e rimuovere i difetti** quando il software non funziona.

10.5 Definizioni di testing

Ci sono differenti definizioni di testing:

- IEEE 829-1983 definisce il testing come il **processo per realizzare un software item** per individuare le **differenze tra existing e required conditions** e per **valutare le features di un software item**
- IEEE 610.12-1990 definisce il testing come il **processo di eseguire un componente sotto specificate condizioni, osservare e memorizzare i risultati e fare delle valutazioni su alcuni aspetti del un sistema o del componente**
- IEEE 829-2008 **definisce il testing come un'attività in cui un sistema o un componente è eseguito sotto specificate condizioni, osservare e memorizzare i risultati e fare delle valutazioni su alcuni aspetti del un sistema o del componente**

10.6 Concetti per testing & debugging



Alcuni concetti sono:

- **Failure:** manifestazione del difetto, evento osservabile percepito dall'utente come una mancata prestazione di un servizio atteso.
- **Fault:** evento che porta alla failure, insieme di informazioni che quando processate producono un fallimento. In sostanza failure è un comportamento anomalo, inatteso o errato del sistema mentre il fault è la sua causa identificata o ipotizzata. Non sempre un fault genera una failure, una failure può essere generata da più fault, un fault può generare diverse failures.
- **Defect:** termine usato per indicare sia la failure che il fault
- **Error:** esso ha due significati:
 - discrepanza tra valore misurato, calcolato o osservato e il valore corretto
 - l'azione di una persona che causa la presenza di un fault in un software

Lo **stato di errore** è lo stato in cui, continuando l'esecuzione, **porterà ad una failure**

10.7 Test e casi di test

Un programma è esercitato da un **caso di test**, ovvero un **insieme di dati di input**. Un **test** è formato da un insieme di casi di test, definito come **test suite**. Un **test ha successo** se rivela uno o più malfunzionamento di un programma.

10.8 Oracolo

L'oracolo è la condizione necessaria per effettuare un test e **conoscere il comportamento atteso** per poterlo confrontare con quello osservato. L'oracolo **conosce il comportamento atteso per ogni caso di test**. L'oracolo può essere:

- **umano**: che si **basa sulle specifiche o sul giudizio**. Quando si inseriscono le **assert in un programma**, l'oracolo è comunque **umano** in quanto l'assert è scritto da noi, ma è **automatico il confronto con l'oracolo**. Si potrebbe avere però un'automatizzazione nella generazione dell'oracolo e relativo controllo
- **automatico**: generato dalle specifiche formali, dalla versione precedente o dal software stesso ma sviluppato da altri. È buona norma avere un oracolo memorizzato per i dati di test, riuscendo ad automatizzare i casi di test

Avere degli **oracoli automatici ai fini dell'esecuzione** è ottimale per **risparmiare tempo**.

10.9 Problemi del testing

Oltre ai **grandi lassi di tempo**, il settore del testing è tormentato da problemi indecidibili, ovvero **problemi per cui non è possibile costruire una macchina di Turing che lo decide**. Se non possiamo dimostrare che un programma si ferma, **tanto meno possiamo dimostrare che un programma è corretto**, poiché se è corretto implica che si ferma.

10.9.1 Problemi del testing: Equivalenza di funzioni

Dati due programmi, il **problema di stabilire se essi calcolano la stessa funzione** è **indecidibile**. Ciò ha avuto enormi conseguenze sul testing poiché dato un programma e supposto noto e disponibile l'**archetipo idealmente corretto di tale programma**, non possiamo comunque dimostrare l'**equivalenza dei due**. Altre indecidibilità derivano da ciò.

10.10 Validazione e verifica: confidenza

Per poter fare validation e verification è **necessario fornire un livello di fiducia che confermi che il software è adatto ai suoi scopi**. Ciò non vuol dire che si ha un software privo di difetti, in quanto è possibile tollerare la presenza di difetti, ma si sta dicendo di avere un software che è abbastanza buono per l'uso per cui è stato pensato.

10.11 Terminazione del testing

Quello che deve fare il testing è quindi **aumentare la fiducia nel programma**. Mi fermo quando il **livello di fiducia è tale per quelli che sono gli obiettivi del programma**. Un programma è testato a sufficienza seguendo:

- **criterio temporale**: periodo di tempo predefinito
- **criterio di costo**: sforzo allocato predefinito
- **criterio di copertura**: percentuale predefinita degli elementi di un modello di programma o legato al criterio di selezione di casi di test. Esso richiede un **testing sistematico**
- **criterio statistico**: non è fatto andando a testare sistematicamente varie parti del programma **ma è fatto su base statistica, ovvero la probabilità di utilizzo di certi dati di test**. I dati di **test più probabili verranno eseguiti più spesso**. Si basa su un **modello operazionale di utilizzo del sistema da parte dell'utente**. Non è detto che si trovino tutte le failures in quanto non viene usato un approccio sistematico. **Ciò che mi interessa statisticamente è il MTBF** (mean time between failures) predefinito e confronto con un modello di affidabilità esistente. **Voglio che questo tempo aumenti. Più aumenta più è affidabile il sistema**

10.12 Test ideale

Un test ideale è quello tale che l'**insuccesso del test implica la correttezza del programma**. Poniamoci di trovare il **test ideale**, ovvero un **insieme di casi di test tale che se tutti i test falliscono per cui non rilevano malfunzionamenti, io posso dire che il programma è corretto**. Un **test esaustivo** contiene tutti i casi di test di un programma, per cui è un **test ideale**. Esso però è **impraticabile** e quasi mai fattibile.

10.13 Criterio di selezione di test

Un criterio di selezione di test **specificava le condizioni che devono essere soddisfatte da un test**. Esso è un criterio che **specificava come selezionare i casi di test**. Un criterio di **selezione di test è affidabile per un programma se per ogni coppia di test selezionati, T₁ e T₂, dove entrambi sono insiemi di casi di test, se T₁ ha successo anche T₂ ha successo e viceversa**. In pratica un criterio di selezione di casi di test è affidabile o se tutti i casi di test generati riescono a rilevare malfunzionamenti o nessuno riesce. L'affidabilità è quindi il fatto che **qualcosa si comporti sempre allo stesso modo**. Per ogni test T_i, dovrà esserci almeno un caso di test che riveli il malfunzionamento, non tutti i casi di test del singolo test T_i.

Un criterio di selezione di test è **valido** per un programma se, qualora il programma non sia corretto, esiste almeno un test selezionabile mediante il criterio di selezione che consente di rilevare malfunzionamento. Il problema di costruire un test ideale è **indecidibile**. Non esiste un algoritmo che, dato un programma arbitrario P, generi un test ideale finito, e cioè un test definito da un criterio affidabile o valido.



10.14 Tecniche di testing

L'obiettivo realistico è di **selezionare casi di test che approssimano un test ideale**. Al di là di casi banali, **non è possibile costruire un criterio di selezione generale di test valido e affidabile che non sia il test esaustivo**. Bisogna quindi cercare di approssimare un test ideale, ad esempio cercando di **massimizzare il numero di malfunzionamenti che si possono rilevare** e cercando di **minimizzare i casi di test**. Spesso, si usa più di un criterio di selezione di test. Alcune tecniche di testing sono:

- **testing dei difetti (sistematico)**: Test progettati per scoprire in maniera sistematica i difetti del sistema, ciò usando tecniche **white box** (testing strutturale) e **black box** (testing funzionale)
- **testing statistico**: Test progettati per riflettere la frequenza di input degli utenti. È usato per la stima dell'affidabilità
- **analisi mutazionale**: consente di valutare la bontà del test.

10.15 Random testing & testing sistematico

Il testing statistico è un testing quasi random. Non è completamente random poichè predilige i casi di test che vengono eseguiti più frequentemente. Si basa quindi sulla frequenza di utilizzo di certi dati di test da parte di un utente. C'è quindi bisogno di costruire un modello dell'utilizzo del sistema da parte dell'utente per generare questi casi di test.

Il testing random (uniforme) è un testing puramente casuale. Non considera la frequenza di utilizzo di certi dati. Ogni dato di test ha la stessa probabilità di essere selezionato. Il testing random andrebbe bene quando gli input hanno tutti la stessa importanza. Ma non è così nella realtà. I casi di test sono eseguiti per controllare il comportamento di un programma, per cui i casi di test sono eseguiti per cercare malfunzionamenti. Io dovrei cercare i test in modo tale che i test selezionati abbiano una maggiore probabilità di malfunzionamenti, per cui utilizzando un random testing si sta dicendo che tutti gli input hanno la stessa probabilità di trovare dei malfunzionamenti, cosa che non è vera.

Si utilizza quindi un approccio sistematico, non uniforme. Non si riesce a sapere quali sono gli input che hanno maggiori probabilità di rilevare malfunzionamenti. L'idea però è quella di partizionare l'input. Si avranno diversi insiemi in cui per ogni insieme il comportamento del programma per gli input dello stesso sottoinsieme è simile, all'interno del sottoinsieme gli input avranno la stessa probabilità di trovare un malfunzionamento, scegliendo poi un rappresentante per ogni classe. Una scelta di questo tipo sarebbe un criterio di testing affidabile e valido. Posso prendere quindi un input a caso per ogni classe, dove la sistematicità nell'approccio sarà nel dividere gli input in classi. Rilevare un malfunzionamento non è uguale per ogni sottoinsieme ma è uguale in ogni sottoinsieme. Il category partition però non è valido o affidabile. I valori dello spazio di input che causano una failure non sono distribuiti uniformemente nello spazio di input per cui io sono in grado di beccarli con un testing random, ma sono bensì sparsi.

10.16 Principio di partizionamento

Nel caso del testing, tutte le tecniche di testing sistematico vanno a partizionare lo spazio di input in sottoinsiemi, in cluster, che conterrà a sua volta casi di test, insieme di input dei valori delle variabili di input. Gli input non sono equiprobabili dal punto di vista di causare delle failures, ma ciò che mi aspetto è che siano densi. È possibile che

alcuni cluster non rilevino malfunzionamenti, mentre altri li rilevano. L'ideale sarebbe di avere o tutte partizioni che non rilevano difetti o avere per ogni cluster un caso di test che individua la failure, avendo un criterio di test affidabile e valido.

Si può partire da una specifica per creare un partizionamento, partendo dalla conoscenza specifica, come effettuato nelle tecniche di testing black box (equivalence class testing o equivalence partition). Alcune tecniche ci danno un **partizionamento totale** di uno spazio di test, per cui i cluster sono disgiunti. Molte altre tecniche ci forniscono un **quasi partitioning**. Non si applica sempre un partitioning esatto poiché più costoso. Partizionando completamente bisogna coprire ogni cluster, mentre con il **quasi partitioning** si possono prendere test che appartengono a più cluster, in quanto l'intersezione tra i vari cluster non è vuota. I criteri sono fatti in modo da avere quelli con il **massimo del partizionamento ma molto costosi**, fino a quelli in cui i cluster hanno più overlap potendo scegliere meno casi di test.

Quello che cerchiamo di fare è **partizionare lo spazio di input ed avere un criterio di copertura**, che fornisce anche un modo di partizionare lo spazio di input e un **criterio di selezione per i casi di test**. Se la rappresentazione su cui ci si basa è **estratta dalla specifica facciamo testing black box**, se **estratta dal codice facciamo testing white box**. Il nostro sistema è fatto di specifica e di software. Facendo testing basato sulla specifica (black box) ci possono essere **funzionalità specificate che non sono implementate**, e ciò può essere individuato solo mediante **testing black box**, in quanto si usa la specifica. Ci possono invece essere parti del sistema **che sono implementate e non specificate**, e ciò può essere trovato solo tramite **testing white box** in quanto si hanno casi di test basati sul codice

Esempio: lo state based testing, ovvero testare un oggetto rappresentato come una macchina a stati finiti, posso selezionare i casi di test andando a coprire i cammini sulla macchina a stati finiti. Se tale macchina è tirata fuori dalla specifica sto facendo black box, se tirata fuori dal codice, cosa fattibile con reverse engineering, sto facendo testing white box, pur avendo stessa tecnica e stesso criterio. Ciò che amba è il come ho rappresentato il software

10.17 Analisi mutazionale

Tecnica che **deriva dal testing dell'hardware**. Essa serve a **generare programmi alternativi**, chiamati mutanti. Si fa testing ma non si rilevano malfunzionamenti. Per valutare se la test suite è buona si va ad inserire difetti nel codice. Se la test suite non è in grado di rilevare malfunzionamenti iniettati, allora la test suite non è buona, per cui deve essere

arricchita con casi di test che vanno a rilevare quel malfunzionamento. Alternativamente, la **test suite è in grado di uccidere il mutante**.

10.18 Testing statistico e affidabilità del software

Usato per il **testing dell'affidabilità del software**. Misurando il numero di errori permette di predire l'affidabilità del software. **Dovrebbe essere prima specificato un livello accettabile di affidabilità e poi il software dovrebbe essere testato e corretto** finché non si raggiunge il **livello di affidabilità desiderato**. Bisogna definire il profilo operativo del sistema e costruire casi di test che riflettono tale profilo



CAPITOLO 11

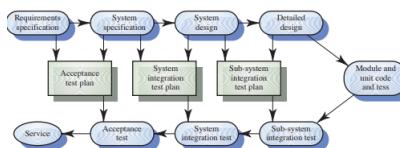
Pianificazione del testing

11.1 Introduzione: pianificazione del testing

Nella pianificazione del testing sono definite le **tecniche per selezionare i casi di test**, dovendo poi **progettare il test che include la specifica di casi di test e la specifica delle procedure del test**. Il test non è una fase ma ha un **insieme di attività** e viene condotto a **diversi livelli**. I test sono parte di un prodotto finale per cui devono essere realizzati con codice, **includendo anche la documentazione** del software usata per fare testing.

11.2 Modello di sviluppo a V

Nelle fasi alte ci sono le **attività di pianificazione del test**. Fasi in cui vengono prodotti **piani di test di accettazione, piani di test di sistema e piani di integrazione del system design**, dove poi questi piani vengono eseguiti nelle parti basse dei processi di sviluppo. Nella parte centrale c'è lo **sviluppo del singolo modulo e il test di unità**. Facendo test **black box** posso sviluppare il codice di test anche prima di aver sviluppato il codice da testare. In caso di **testing white box**, i test devo svilupparli dopo aver progettato la **logica interna del modulo** per capire quale parte di codice andare ad eseguire con i test. Le fasi sono: **identificazione dei requisiti, specifica dei requisiti, progetto di alto livello, progetto di dettaglio, codifica e test di unità, integrazione e test di sistema, accettazione e rilascio, operatività e manutenzione**

**Figura 11.1:** Verifica e convalida

Lo sviluppatore del singolo modulo dovrebbe sviluppare il singolo modulo, quindi dovrebbe fare **testing di unità**. Dovendo fare testing di **integrazione e di sistema** è necessario **avere un team di sviluppatori esterni**.

11.3 Livelli di testing

I livelli di testing sono: **testing di unità**, **testing di integrazione** e **testing di sistema**. Il cliente ha l'onere di fare **test di accettazione**. Esistono test che vengono effettuati con la **collaborazione tra produttore e cliente**, come l'alpha e beta testing:

- La **versione beta** del software viene **rilasciata nell'ambiente del cliente**. Lo **sviluppatore non ha controllo di cosa accade**
- La **versione alpha** viene **rilasciata sotto il controllo dell'azienda**, non necessariamente all'interno dell'azienda stessa. Si può **monitorare l'esecuzione del sistema**, vedere cosa succede e **poter intervenire**. Ciò non può essere fatto nella versione beta

11.4 Testing di unità

Il testing è applicato **isolatamente ad una unità** (modulo) di un sistema software. L'obiettivo fondamentale è quello di **rilevare errori** (logica e dati) **nel modulo**. La prassi diffusa è che venga **realizzato direttamente dal programmatore che ha sviluppato il modulo**, l'unità sottoposta a test. Per **unità si intende un elemento definito nel progetto di un sistema software e testabile separatamente**. Unità e modulo sono spesso usati come sinonimi. Nel caso in cui l'unità è chiamata da un modulo è necessario un **test driver**, mentre nel caso in cui l'unità usa un modulo che ancora non è disponibile è necessario uno **stub**. Nella programmazione ad oggetti il **concepto di unità è la classe**, testabile separatamente. L'**output** del programma deve essere poi **confrontato con un oracolo**.

11.5 Problema dello scaffolding

è necessario creare l'ambiente per l'esecuzione dei test. Lo **scaffolding** è estremamente importante per il **test di unità** ma anche per il **test di integrazione**. Può richiedere un grosso sforzo programmatico. Uno **scaffolding buono** è un passo importante per **test di regressione efficiente**. La generazione di scaffolding può essere parzialmente automatizzato a partire dalle specifiche. La necessità di **driver** e **stub** dipende dalla posizione del modulo rispetto l'**architettura del sistema**. Se un modulo si trova sul fondo dell'architettura, ovvero non usa nessun altro modulo, non si avrà bisogno di stub.

11.6 Generazione test driver e test stub

La generazione di driver è stub può essere:

- **interattivo:** richiesto l'intervento umano, ma **tropo oneroso**. L'utente potrebbe introdurre errori. Un driver interattivo è un **driver che chiede in input** dei valori all'utente, mentre ad uno **stub interattivo viene passato un valore** e, mostrandolo all'utente, **chiede a quest'ultimo il risultato della funzione**.
- **automatici:** in tal caso:
 - **driver:** dovrebbe avere un **file con i dati da cui prende gli input**, o potrebbe inizializzare l'ambiente non locale. Facendo category partition, si hanno degli oggetti dell'ambiente non locale alla funzione che si sta testando. Essi sono un qualcosa con cui la **funzione che stiamo usando interagisce ma che non possiamo fornire come input**. Essi però andranno ad influenzare il risultato del test
 - **stub:** **Calcolare valore approssimato o la restituzione di valori costanti**. A volte però non basta. Dipende da cosa stiamo testando. Se ad esempio la funzione dipende dal valore risultato dello stub, conviene calcolare il valore e non fornirlo costante

C'è un **trade off tra i costi di progettazione del test, sviluppo di driver e stub ed i costi di esecuzione del test di regressione**. Più spendo in costi di sviluppo, meno spendo in costi di esecuzione.

11.6.1 Software OO e Unit Testing

All'interno del software OO, una possibile suddivisione nei testing è:

- **Basic unit testing:** test di una **singola operazione di una classe** (intra-method testing)
- **Unit testing:** test di una **classe nella sua globalità** (intra-class testing)

è più complicato fare testing con questi software poiché nei linguaggi procedurali standard la componente base è la procedura e il metodo di test si basa sull'input e sull'output. Per quanto riguarda OO è necessario fare **differenziazione tra testing dei singoli metodi e testing della classe**. Nel caso dei SOO sarà necessario avere **test cases che testano non solo i metodi ma anche il comportamento della classe**. Tale testing verrà fatto inviando una serie di eventi alla classe. L'ordine di invio dovrebbe esercitare un particolare comportamento della classe. Queste tecniche **si basano su un modello di rappresentazione del comportamento della classe che si basa su statechart diagram**. Esso prende il nome di **state base testing**. Ogni messaggio che l'oggetto riceve dovrebbe provocare una **transizione di stato**, per cui si andrà a verificare che le **transazioni di stato** che vengono effettuate al susseguirsi dell'invio di messaggi, **corrisponde al comportamento atteso**.

11.6.2 Information hiding

Gli oggetti sono istanze di classi e **la correttezza non è legata solo all'output**, ma anche allo **stato della classe**, definito dalla struttura dati. Lo stato della classe è però **privato** e può essere osservato solo utilizzando **metodi pubblici**, che potrebbero essere sbagliati. Il **driver** deve settare le variabili di stato usando i metodi **set** e esaminare lo stato per l'**oracolo**, che conterrà non solo l'output del metodo ma anche lo **stato in cui deve trovarsi la classe**. Ci sono **approcci intrusivi che vanno a modificare il codice sorgente** o usare i costrutti del linguaggio (come il costrutto friend). Una **classe friend** può accedere alle variabili di istanza di un'altra classe.

11.7 Testing di integrazione

Il testing di **integrazione** viene applicato a un aggregato di uno o più unità di un sistema software. Il testing di integrazione deve essere eseguito dopo il **testing di unità**, per cui si va ad evidenziare i problemi afferenti a classi differenti che collaborano. Ci possono essere diversi errori di integrazione:

- **interpretation error:** quando specifichiamo l'interfaccia di una classe, noi **stiamo definendo una sorta di contratto** che deve essere rispettato sia da chi usa la classe, sia chi la implementa. **Se chi la usa e la implementa capiscono cose differenti della specifica**

si avrà un **interpretation error**. In questo caso sarà necessario capire in quale delle due classi risiede l'errore

- **miscoded call error:** chiamata inserita nel posto sbagliato o dove non viene richiesta. Questi sono più **errori di logica**
- **interface error:** si ha una **violazione dell'interfaccia standard tra due moduli**. Esso è un errore che si può avere molto spesso in linguaggi in cui non ci sono controlli formali tra **corrispondenza fra parametri formali e parametri attuali** (compito del sistema operativo accorgersene).
- uso di **drivers/stub poveri** durante il test di unità

11.7.1 Modalità di testing di integrazione

Esistono diverse tecniche per l'integrazione, come:

- **big bang testing:** tutto viene testato insieme, avendo però **problemi in fase di localizzazione del fault**. Nel caso di SOO non è possibile applicare il big bang
- **tecniche incrementali:** integrazione dei moduli man mano che vengono prodotti e testati.

Il vantaggio è di limitare l'uso di **moduli driver e stub** (moduli fittizi e guida che chiamano o vengono chiamate). Testando incrementalmente si avrà una **maggior facilità nel capire dove siano allocati gli errori**. Ci saranno, inoltre, **moduli che verranno esercitati più a lungo**, perché verranno eseguiti con test diversi, avendo una **maggior probabilità di trovare malfunzionamenti** che sono allocati all'interno di un particolare modulo e che non ho rilevato con il test di unità. Nel caso di **SOO per bottom up e top down**, non si parla solo della **dipendenza di uso**, come nei software procedurali, ma si parla anche di **dipendenza dinamica**, in quanto il metodo di una classe può invocare anche il metodo di un'altra classe. Si parla inoltre di **ereditarietà**, ovvero se **A eredita da B, A dipende da B in quanto se B cambia anche A deve cambiare**. È preferibile in questi casi una strategia **bottom up**, testando **prima le classi indipendenti**. Per l'**ereditarietà è impossibile pensare di usare un approccio diverso dal bottom up**, a prescindere da se la classe dipende è sopra o sotto la classe da cui dipende. Bisognerebbe alternativamente costruire uno **stub per la superclasse**, il che non ha molto senso.

11.7.2 Top down

Si parte dai moduli in alto e li va ad integrare, per cui non si ha bisogno di moduli driver. Si può esplorare in ampiezza o in profondità, l'importante però è che andando al livello successivo i moduli del livello precedenti siano già testati. Inoltre, si riuseranno sempre gli stessi test, questo perchè i test rimarranno gli stessi propagandosi verso il basso. Ciò potrebbe essere anche un problema in quanto i moduli a livelli alti verranno testati sempre con gli stessi dati. In questo caso si avranno però stub molto complessi da realizzare. Gli stub facili da realizzare sono quelli che non impattano sull'esecuzione. I moduli di ingresso/uscita ed i moduli di elaborazione sono postposti.

11.7.3 Bottom up

Si testano prima i moduli ai livelli più bassi. Si vanno a postoporre i moduli che gestiscono la logica di controllo. Se ho un problema a livello architettonale è nella logica di controllo che si andrà a manifestare l'errore. Sono richiesti moduli driver piuttosto che moduli stub. La logica di controllo è sempre in alto. Con queste tecniche si hanno differenti dipendenza come uso di classi ed ereditarietà. È preferibile quindi una strategia bottom up, andando a testare le classi indipendenti. In questo caso i test cambiano per ogni modulo che vado ad integrare, avendo il vantaggio di avere più esecuzioni dei moduli più in basso con test case differenti.

11.7.4 Sandwitch

Testing di integrazione che prevede l'unione di testing bottom up e top down. Il vantaggio che si andranno a creare meno test stub e test drivers. Partiamo dal livello centrale. Vorremmo ridurre al minimo il carico di lavoro. In questo caso, eseguiamo alcuni test unitari ai livelli inferiori, quindi applichiamo test di integrazione e dopo testiamo la piena integrazione (in un sistema a 3 livelli). Partiamo dalle unità e dopo procediamo. Mescoliamo le varianti di bottom up e top down. Non applichiamo mai unit test dei componenti intermedi. Se testiamo anche i componenti del livello centrale, abbiamo un modified sandwich, una delle tecniche più potenti e che richiede più tempo. Con modified sandwich **bottom up e top down sono approcci condotti in parallelo** e tutti i componenti vengono prima testati individualmente.

11.7.5 Grafo delle dipendenze

Grafo usato per **rappresentare dipendenze tra le classi**. Se il **grafo è aciclico** esiste un **ordinamento parziale sui suoi elementi**. Il < indica chi deve essere testato prima. In foto si fa riferimento all'ordinamento totale topologico.

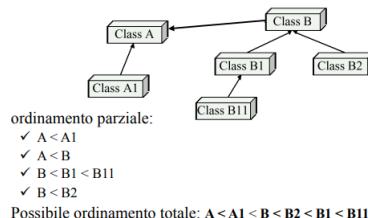


Figura 11.2: Verifica e convalida

Se si ha un **ordinamento totale**, dati due qualunque elementi dell'insieme, **è possibile decidere quale sia il più piccolo e quale sia il più grande**. Nell'ordinamento parziale ciò non è possibile. Tuttavia, **dati due elementi di un insieme è possibile trovare un terzo elemento più piccolo o grande di tutti e due**. L'ordinamento parziale si ha su una struttura gerarchica, come una struttura ad albero. Si sa chi è il figlio di chi, ma dati due elementi qualsiasi dell'albero non si sa dire chi dei due sia maggiore. Dato un **ordinamento parziale**, è possibile trovare un ordinamento totale topologico. Facendo la visita di un albero, si fa la visita di una struttura che mantiene l'ordinamento parziale. Nel momento in cui si decide in che ordine visitare gli elementi, si andrà in ogni caso andare a visitare prima un elemento più in alto e poi l'elemento più in basso, che sia mediante BFS o DFS. **L'ordinamento totale topologico è un ordinamento totale degli elementi che rispetta l'ordinamento parziale**, ovvero le relazioni d'ordine vengono rispettate. Per fare ciò si vanno a privilegiare le **precedenze di specializzazione**, testando prima le **gerarchie di classe** e poi testando le **interazioni tra gerarchie**. Se **esistono dipendenze cicliche**, allora diventa più **difficile definire un ordinamento parziale**. È inoltre possibile integrare classi in cluster, mantenendo però l'ordinamento parziale.

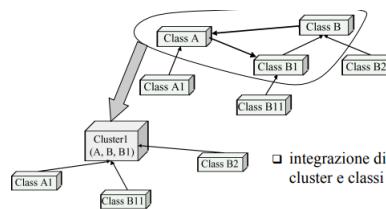


Figura 11.3: Verifica e convalida

Esempio: L'insieme delle parti di un insieme P, con operazioni unione, intersezione e

complemento, costituisce un'algebra di bool. Valgono tutte le proprietà ed è un reticolo distributivo dotato di minimo e massimo, ovvero l'insieme vuoto e l'insieme P stesso. Anche sul reticolo si ha un ordinamento parziale. Per privilegiare le precedenze di specializzazione, si va a testare prima le classi in B e poi in A. Si potrebbe infatti testare A1 prima di tutte, poiché varrebbe l'ordinamento totale topologico, ma facendo così non si rispetterebbero le precedenze di specializzazione in quanto B che dipende da A verrebbe testata dopo A1, da cui però non dipende ma fa parte della gerarchia di A.

11.7.6 Problemi di integrazione per SOO

Una volta definito l'ordine di integrazione si aggiungono le classi incrementalmente esercitandone le interazioni. Possibili problemi di integrazione:

- **Ereditarietà:** implica problemi in caso di modifiche di superclassi
- **Polimorfismo:** comporta problemi legati al binding dinamico

11.8 Testing di sistema

è il testing applicato sul sistema software completo ed integrato. Si va a valutare l'adesione del sistema rispetto ai requisiti specificati e va eseguito dal team addetto al testing esterno al team di sviluppo. Non si vanno a valutare requisiti funzionali basati sul caso d'uso, ma anche non funzionali come testing di affidabilità, robustezza, sicurezza, configurazione (si va a vedere se cambiando le configurazioni continua a funzionare tutto come si deve).

11.9 Test di accettazione

Viene fatto a carico del cliente. Essa è più una demo che un test in quanto l'obiettivo è convalidare il prodotto piuttosto che trovare errori.

11.10 Alpha e beta testing

Si hanno due tipi di testing:

- **Alpha Testing:** Uso del sistema da parte di utenti reali ma nell'ambiente di produzione e prima della immissione sul mercato. Talvolta riferito per il testing da parte di un cliente (o gruppo di clienti) privilegiato

- **Beta Testing:** Installazione ed uso del sistema in ambiente reale e prima della immisione sul mercato. Strategia adottata da produttori di packages per mercato di massa. Talvolta il **beta testing** è preceduto da un **alpha testing**, o un **beta testing** da parte di **un gruppo più ristretto di utenti**. Si hanno problemi di confidenzialità



CAPITOLO 12

Documenti di testing

12.1 Introduzione: documenti di pianificazione

Oltre al MTP, ci sono i **test plan** dei vari livelli. Per ogni livello, oltre al test plan, c'è il **test design, test case e test procedure specification**. Alcune definizioni sono:

- **Test item:** ogni oggetto che sottopongo al testing. Esso può essere codice sorgente, codice oggetto, job control, data control... ed è accompagnato da relativa documentazione. Chi deve eseguire il test deve anche specificare il test
- **Pass/fail criteria:** regole di decisione da usare per stabilire se una caratteristica software supera o meno il test. Essi dipendono dal tipo di testing che si sta facendo

12.2 Documenti di pianificazione e specifica

12.2.1 Master Test Plan (MTP)

Documento generale per la pianificazione e gestione del testing. Esso contiene il piano di progetto relativo a tutte le attività di testing. In questo piano è quindi specificato quando vengono svolte determinate attività di testing. Avrò varie attività che potranno essere decomposte in sottoattività, facendo una suddivisione delle attività anche in base al livello di testing, contenendo inoltre le attività di pianificazione di questi testi. Nell'MTP è quindi

indicato tutto, quando vengono svolte le varie attività, con quale tempistica, quali sono le risorse coinvolte...

è quindi un piano di progetto relativo alle attività di testing. Si definiranno anche **obiettivi dei test, tecniche utilizzate...** andando quindi a dettagliare delle cose definite nel piano di progetto dove però vengono solo nominate. Per ogni livello di testing viene poi fatto un **level test plan**, avendo quindi livello del singolo componente fino a test di accettazione

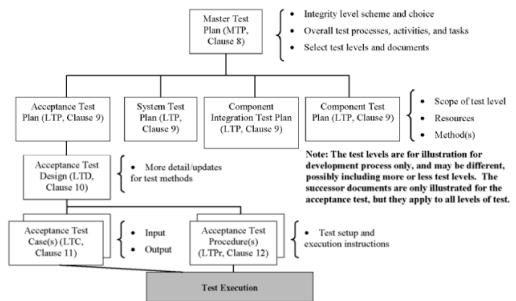


Figura 12.1: Verifica e convalida

Si avrà quindi un MTP, con test plan per vari livelli. **Ognuno di essi avrà un test design che include i casi di test e le procedure.** Si possono avere varie test suite e per ognuna di esse è necessario definire un LTC e un level test procedure. I documenti di specifica dei casi di test andranno agli sviluppatori che devono eseguire il test. Se si ha un sistema molto grande e si è distribuito le attività di system design a più persone, **ognuno dovrà avere il documento che gli serve**, per cui **sulla base della pianificazione si andranno a costruire documenti** che andrò ad assegnare a risorse diverse che vanno ad eseguire il test.

12.2.2 Level test plan

In questo documento **vengono individuati i test item**, in quanto i test item **cambiano in base al livello del test**. È inoltre necessario **definire le attività che devono essere fatte**, come **le attività si decompongono**, quali sono le **risorse che lavorano all'attività**, quali sono i **rischi**. È necessario inoltre **identificare i pass/fail criteria ed identificare i deriverables delle attività di testing**.

12.2.3 Level test deisng

Documento che specifica per una o più combinazioni di caratteristiche da testare i dettagli dell'approccio al testing. In questa fase si vanno ad **inserire i dettagli**, mentre nel piano di test era solo fornita una linea guida generale. Come dettaglio si intende **anche il procedimento**

stesso. A seconda di quanto è **complesso il test di sistema che vado a fare**, posso avere **più documenti di design**. Nel caso in cui il **test di design sia molto complesso** esso è da **tenere separato rispetto all'LTP**, in alternativa, invece, **esso può essere integrato nell'LTP**. L'idea di separarli nel caso di complessità maggiore dipende dal fatto che **approcci e tecniche utilizzate e che vanno spiegate all'interno del documento sono diverse**. Nell'LTP si farà quindi riferimento ai vari LTD. Il test design dovrebbe **includere i casi di test**. È necessario partire dai **casi d'uso del sistema** in quanto ogni caso d'uso avrà la sua **test suite**. È possibile quindi inserire il procedimento che mi ha portato ai casi di test che vengono dopo, ma è poi necessario **Mantenere tale tracciabilità tra il design del test ed i casi di test**, se vengono messi in due documenti a parte. Nel caso in cui vengano messi nello stesso documento non c'è problema. Si avrà quindi un maggior livello di coesione tra LTD e LTC che non tra LTP e LTD.

12.2.4 Level test case

Definito anche come **test case specification**, esso specifica **tutti i casi di test individuati dal segmento di LTD associato**. Per ogni test case viene specificato:

- **test case identifier:** Il test case identifier sarà lo stesso presente nell'LTD, in modo quindi da mantenere la tracciabilità. Passando da LTC ad LTD usando l'identifier posso capire come ho ricavato il caso di test, capire quale sia il test frame di riferimento
- **obiettivo:** obiettivo del test case
- **input:** input del test case
- **oracolo:** oracolo del test case
- **condizioni di esecuzione:** chi deve eseguire il test **deve sapere come eseguire il test**. È necessario quindi non solo il caso di test ma anche la procedura di test (passi da eseguire per effettuare il test)

È necessario **Mantenere la tracciabilità tra i test frame e i casi di test**. Andando a **modificare il modo in cui si costruisco i test frame**, cambieranno anche i casi di test. Senza questo collegamento, **in caso di modifica al test frame non si riesce a capire quali test bisogna sostituire e quali aggiungere**.

12.2.5 Level test procedure

Documento che specifica per uno o più test case i passi da fare per eseguirli, in particolare:

- come preparare l'esecuzione della procedura
- come avviare e condurre tale esecuzione
- quali rilevazioni e misure fare
- come sospendere il test in presenza di eventi imprevisti
- come riprendere un'esecuzione sospesa

Se eseguo il test e mi devo fermare, è necessario definire anche come riprendere il test. Ciò potrebbe essere utile nel caso di test che hanno una durata lunga. Una test procedure è quindi una sequenza di istruzioni dettagliate per il set-up, esecuzione e valutazione dei risultati. È possibile avere una stessa procedura che si applica a più test suite, infatti di solito questi test sono in numero minore rispetto ai LTC. Al più saranno uguali, non avendo LTC con più procedure

12.3 Documenti di esecuzione

12.3.1 Level Test Log (TL)

È una banca dati della memorizzazione sistematica, strutturata ed in ordine cronologico di tutti i dettagli rilevanti sulla esecuzione dei test. Tutte le esecuzioni di test sono memorizzate nel TL. Ogni test eseguito viene memorizzato. Informazioni fondamentali memorizzati sono il successo o l'insuccesso dei test, chi ha eseguito il test, a che ora è stato fatto... In esso è presente anche la descrizione di eventi anomalie.

12.3.2 Anomaly report (TIR)

Documento che descrive ogni evento occorso in un processo di testing e che richiede altre e più approfondite analisi ed investigazione. Nel caso della manutenzione il TIR contiene anomalie rilevate dall'utente, non risultato del testing quindi ma risultato dell'esecuzione del sistema, ovvero qualcosa successo durante l'esercizio. Tali tipi di anomalie però si possono avere anche durante il testing. Chi esegue il testing, in presenza di un'anomalia, va a segnalarla. Nel TIR sono inclusi: gli input, i risultati attesi, i risultati attuali, anomalie,

data ed ora, le azioni correttive intraprese ed i tentativi di rieseguire il test. Si vanno a salvare sia i tentativi ma soprattutto le azioni correttive poichè, nel caso il problema si ripresenti successivamente, si potrà andare a vedere nel TIR come tale problema sia stato risolto precedentemente. Una delle **misure di affidabilità che si fa è proprio calcolare il numero di malfunzionamento in un certo tempo**, o volendo il MTBF. Il **numero di malfunzionamenti è un qualcosa che misura il livello di affidabilità in un processo di testing.** Il numero di malfunzionamenti dipende dal software, ma non solo, in quanto **dipende anche da chi ha pianificato il test, come è stato pianificato, chi esegue il test.** Il **numero di malfunzionamenti è quindi una misura diretta del processo di test** che uso come misura indiretta di un attributo del prodotto

Esempio: Nel caso ci sia un problema e rieseguo il test. Nel test log ci saranno entrambe le esecuzioni del test fallito. Il test viene poi corretto e tutte queste informazioni vengono salvate

12.4 Level Interim Test Status Report (LITSR)

Esso è un **documento per ogni livello e riassume lo stato del testing rispetto ai piani.** Descrive i **cambiamenti rispetto ai piani e descrive le metriche raccolte.** Tale report serve a dire **quanto è stato fatto rispetto a quanto è stato pianificato.** Questo è un report di management, un report che serve a **capire lo stato di avanzamento.**

12.5 Level Test Report (LTR)

Individua gli **item testati indicandone la versione e l'ambiente in cui sono svolte le attività.** Fornisce i **dettagli dei risultati di test, individua anomalie risolte e ne riassume le soluzioni,** individua le anomalie irrisolte. Ciò è noto anche come **test execution report.** Mentre nel LTR per ogni oggetto si indica cosa è successo, nel TL è **presente un'ordine cronologico di ciò che è successo.**

Se un **qualcosa è stato quindi testato due volte** in quanto si è riparato un problema creato durante l'esecuzione del test, **si avranno due esecuzioni diverse**, un TIR che riporta le due esecuzioni e nell'LTR indicherò per quel test item cosa è successo. Mentre nel TL si ha l'ordine cronologico, **nell'LTR si avrà un'unica riga che indicherà quante volte il test è stato eseguito e cosa è successo le varie volte**, facendo riferimento al TIR. L'LTR riassume quindi **per ogni test item le attività di testing che sono poi documentate cronologicamente nel TL,**

indicando le anomalie dei test item presenti nel TIR. Tale documento riporta le **variazioni dei test items dalle loro specifiche e le variazioni rispetto dalla documentazione di testing e le relative motivazioni**. Specifica inoltre eventuali **issues emerse** durante l'esecuzione del testing e **riporta il ragionale delle decisioni prese**. Fornisce una valutazione delle attività di testing e ne individua le limitazioni. Per **comprehensiveness si intende una valutazione di quanto il test sia esaustivo rispetto agli obiettivi**.

12.6 Master Test Report (MTR)

Fa la sintesi di tutte le attività di test. Nel caso in cui si abbia un'organizzazione articolata per le attività di test, il responsabile delle attività di testing, è colui che prende il piano di test complessivo, il MTP, e lo da al project manager. Il project manager dovrà sapere qual'è il piano di test complessivo e il report master, ovvero documento in cui sono contenuti i problemi, quindi MTP ed MTR. Non sarà necessario per il porject manager conoscere tutti i dettagli delle esecuzioni dei vari test cases. Il responsabile dell'attività di testing, invece, avrà da comunicare con il responsabile per il test di sistema, responsabile per il test di integrazione...

L'MTR fornisce una valutazione complessiva, descrive lessons learned e fornisce conclusioni e raccomandazioni relative all'accettazione del prodotto. Specifica ragioni di decisioni di pass/fail o conditional pass, ovvero un pass sotto specifiche restrizioni di uso basato sulla riduzione di un sottoinsieme di anomalie.

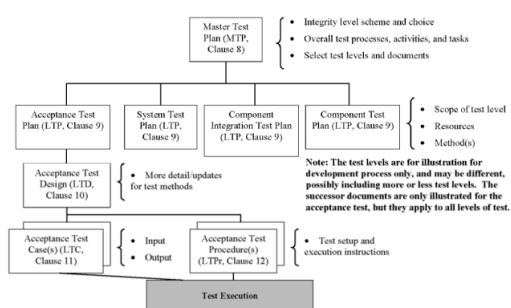


Figura 12.2: Verifica e convalida

Si avrà un **documento che avrà a che fare con lo stato di avanzamento**, ovvero il LITSR, mentre si avranno documenti di level test report che avranno a che fare con il singolo log suddivisi però sulla base del livello testato. L'MTR farà riferimento ai vari level test report.

CAPITOLO 13

Testing funzionale o testing black box

13.1 Introduzione: Testing funzionale

Il testing funzionale è una tecnica di testing in cui la definizione dei casi di test e dell'oracolo è fondata sulla base della sola conoscenza dei requisiti specifici del sistema e dei suoi comportamenti, usando esclusivamente la conoscenza della specifica. Se sta facendo testing di sistema, userò la conoscenza dei requisiti funzionali o dei casi d'uso. Se si sta facendo testing di unità, invece, userò la conoscenza o la specifica dei moduli, come pre-condizioni e post-condizioni. Le stesse tecniche black box a livello di unità possono essere applicate a livello di sistema.

13.2 Black box & white box: differenze

La differenza tra white e black box testing sta nella pianificazione e non nell'esecuzione, in quanto è nella pianificazione che si usano tecniche white e black box. Nello specifico, si ha una differenziazione nella fase di test design e test case specification. Facendo testing white box non guardo il codice durante l'esecuzione, ma il codice vado a guardarla prima, quando scelgo i casi di test. Nel white box si usa la conoscenza della struttura del codice per scegliere i casi di test, usando certe tecniche. Nel caso di black box i test che si generano a livello di sistema e a livello di caso d'uso sono gli stessi. Non è quindi detto che si riescono a testare tutte le interazioni del codice. Non è possibile usare quindi solo test black box.

A livello di sistema il testing black box fa raggiungere una copertura inferiore rispetto a testing black box a livello di unità.

13.3 Scalabilità del black box

Le tecniche black box sono **tecniche sono scalabili**, potendole usare sia a livello di unità che a livello di sistema. Esse sono **scalabili in quanto non dipendono dal codice specifico**, come invece accade per il testing white box. Scalabile vuol dire che **quello che trovo a livello di funzionalità interna come specifica, è molto simile a quello che trovo come funzionalità esterna**. Anche se il comportamento esterno è dovuto da più funzioni, comunque non si ha il problema di scalabilità. Proprio per questo motivo, **soprattutto per il livello di sistema e non a livello di unità**, quello che si fa è **valutare la copertura white box e invece selezionare i casi di test con tecniche black box**. A livello di unità si può anche pensare di usare **criteri di selezione white box**.

Nel caso si consideri una funzione, per fare unit testing, il codice è abbastanza contenuto. Se invece **se si considera tutto il sistema percorsi interprocedurali sono molto più articolati**. È molto più difficile **capire quale sia l'input che devo usare per andare ad eseguire un certo percorso se sto a livello interprocedurale**. Dire **white box** significa dire che vedo il codice, vedo i percorsi di esecuzione e scelgo quali percorsi voglio eseguire, trovando degli input che mi servono per quel percorso. Se lo faccio **a livello di unità** è più semplice, se lo faccio a livello di sistema diventa impraticabile. **Andando ad eseguire dei percorsi a livello di sistema e non riuscendo a raggiungere dei punti, non si sa se l'input non sia stato in grado di raggiungere quel punto poiché il test scelto è insufficiente o perchè quello sia un unfeasible path**. Nel caso di testing white box si utilizzerà la conoscenza relativa alla struttura del programma.

13.4 Riuso black box & white box

Quando si parla di **riuso black box** si intende che si utilizza una componente conoscendo **la specifica senza conoscere o modificare il contenuto**. Quando si parla di **riuso white box** si intende che noi conosciamo il codice e quindi possiamo riusare il codice andandolo a modificare.

13.5 Tecniche di testing funzionale

le tecniche di testing funzionale sono:

- **funzionalità esterne:** funzionalità visibili all’utente e definite dai requisiti e dalle specifiche. Testiamo funzionalità esterne se stiamo a livello di sistema
- **funzionalità interne:** funzionalità non visibili all’utente e definite dal progetto di alto e basso livello. Testiamo funzionalità interne se siamo a livello di unità

I metodi che possono essere utilizzati per l’analisi delle funzionalità sono:

- **basati su specifiche formali:** usando specifiche formali posso generare automaticamente almeno casi di test che identificano le combinazioni di classi di equivalenza che devono essere testate
- **baseate su specifiche semi-formali**
- **basati su specifiche informali:** tecniche manuali almeno fino all’individuazione delle combinazioni di classi



13.6 Criteri di copertura per testing funzionale

I criteri sono:

- **eseguire almeno una volta ogni funzionalità.** Esso può essere un criterio di copertura.
- Per ogni funzionalità posso decidere quali esecuzioni andare a fare, quanti casi di test selezionare. Ciò può essere dedotto dalla specifica, dai dati di ingresso, precondizioni e postcondizioni.

13.7 Suddivisione in classi di equivalenza

Sia testing black box che white box vogliono ottenere un partizionamento dello spazio input che ci consente di avere, preferibilmente, cluster di input che, o tutti individuano un malfunzionamento o nessuno di essi lo individuano. Realizzare un criterio di selezione di casi di test che sia affidabile e valido è qualcosa che non è possibile ottenere (problema indecidibile), per cui si cerca di approssimare. Con il problema delle classi di equivalenza si usa la specifica per partizionare ogni dominio dei parametri input in classi di equivalenza, andandole poi a combinare.

Il dominio dei dati in ingresso è **partizionato in classi di dati di test in modo tale che, se il programma è corretto per un caso di test, si possa dedurre ragionevolmente che è corretto per ogni caso di test in quella classe**. La decomposizione in classi di equivalenza viene fatto sulla base della specifica del problema. In base a come ho specificato il problema, così avrò i casi di test. **Se la specifica è fatta bene si avranno casi di test migliori.** Io mi aspetto che il **comportamento del programma** per tutti i dati di input all'interno di quella classe di equivalenza, **sia lo stesso.** È così solo se il **risultato è indipendente dall'interazione di classi di equivalenza**, ma spesso non è così.

Le **classi di equivalenza** possono essere **valide e non valide**, dove **non valide** sono le **classi che non rispettano le precondizioni**. Tali classi di equivalenza sono utilizzati per **effettuare test di robustezza**. Ogni **combinazione di classi di equivalenza** mi dà una **partizione dello spazio di input**. A seconda di quale tecnica di equivalence partition andiamo ad utilizzare, il **partizionamento sarà esatto oppure no**. Il concetto è che la tecnica di equivalence partitioning va a partizionare in classi il **dominio dello spazio di input**. Non bisogna però testare un singolo parametro ma bensì una **combinazione di parametri** per cui è necessario **combinare ogni classe con le altre per avere un input**. È poi necessario scegliere un criterio di selezione di casi di test

Esempio: nel caso del fattoriale si analizza il caso 0, caso limite e caso 1, avendo tre classi di equivalenza.

13.8 Criteri di copertura deboli e forti

Bisogna partizionare lo spazio di input. La **classe di equivalenza non è una partizione dello spazio di input**. Lo è solo solo se si ha un'unica variabile di input. Se lo spazio di input è formato da più variabili, ogni classe sarà formata da più valori, una per ogni variabile, per cui è necessario incrociare i valori, andando a coprire interazioni di classi di equivalenza. Esistono due tipi di criteri di selezione di casi di test, basati sulla **tecnica delle classi di equivalenza**, e sono:

- **weak:** per coprire tutte le classi di equivalenza si sceglie un valore che copre classi di equivalenza di più variabili. Prendo quindi un caso di test che **copre contemporaneamente classi di equivalenza di più variabili**. Un singolo caso di test andrà a coprire più classi di equivalenza. Il numero di test case che mi servono nel **caso minimo del Weak Equivalent Class Testing (WECT)** è **pari al numero massimo di classi di equivalenza in cui è partizionato il dominio**. Si potrebbero prendere anche più casi

di test, ma dal momento in cui si cerca di ridurre il numero di casi di test, **si cerca di coprire sempre classi differenti**, il che porta a minimizzare i casi di test scelti. In questo caso si ha un **quasi partizionamento**, in quanto si hanno **input che possono ricadere in cluster differenti**. Questi saranno gli input presi maggiormente in considerazione. **Ogni classe di equivalenza deve essere coperta almeno una volta** ma posso prendere casi di test che coprono contemporaneamente più classi di test

- **strong: vado a coprire tutte le possibili interazioni tra le classi.** Si ottiene un **partizionamento perfetto**, poichè un input che ricade in un cluster non ricade in un altro cluster. **Esso richiede troppi casi di test.** Per fare **Strong Equivalent Class Testing (SECT)** è pari alla cardinalità di classe di test moltiplicata per le altre. In questo caso si ha un partizionamento esatto.

Più perfetto è il partizionamento più sono i casi di test che mi servono. Se il criterio partiziona lo spazio di input in maniera perfetta si necessiteranno più casi di test, rispetto ad un partizionamento quasi perfetto. Più è accurata la specifica, migliore sarà la suddivisione delle classi di test. Se le condizioni di errore hanno alta priorità, bisogna estendere il criterio forte (SECT) includendo anche classi non valide. Il metodo delle classi di equivalenza è appropriato quando ragiona su domini discreti, intervalli e insiemi di valori. Il criterio forte (SECT) assume che le variabili sono indipendenti. Le dipendenze tra le classi possono generare "error" test case. È possibile avere classi valide la cui combinazione è una classe non valida, che non rispetta le precondizioni. Con equivalent class testing non si può separare nettamente il test di robustezza dal test di correttezza, questo perché parti del test di robustezza finiscono le test di correttezza.

13.9 Selezione delle classi di equivalenza

Per la selezione delle classi di equivalenza, se la condizione sulle variabili di ingresso specifica un intervallo di valori, allora devo prendere almeno una classe valida per l'intervallo di valori, ed una classe non valida al di sopra ed al di sotto dell'intervallo. L'intervallo valido può essere formata da una o più classi di equivalenza. Mentre la precondizione mi serve per distinguere la classe di equivalenza valida che eventualmente posso partizionare in più classi di equivalenza, indicando anche le classi non valide, la postcondizione è quella che mi va a partizionare l'intervallo in classi di equivalenza. La postcondizione mi individua il modo di dividere in classi di equivalenza.

Nella WECT ogni classe di equivalenza dovrà essere coperta da almeno un caso di test. **Dobbiamo comprendere il maggior numero di classi valide non ancora scoperte**, ovvero cercare di selezionare sempre classi non ancora testate, e si usa un caso di test per ogni classe non valida. Non si può usare un caso di test per più classi non valide in quanto le classi non valide sono la rappresentazione di violazioni delle precondizioni. Quando si testa una classe non valida, tutte le altre devono essere valide. Alternativamente, nel caso si usino più classi non valide, non si sa quale sia la precondizione violata.

L'efficienza delle classi di equivalenza dipende da come sono state selezionate le classi di test. Se sto ragionando in termini di range e intervalli, è difficile che sia ottimale con le classi di test, in quanto, seppur partizionando l'intervallo in classi di equivalenza, l'estremo è incluso nell'intervallo. Il modo di partizionare l'intervallo in classi di equivalenza mi porta ad avere dei sub-range. Per ogni sub-range il comportamento del programma è sempre quello. Ma potrebbe esserci il caso in cui si siano implementate male un test che controlla che un valore ricade nel sub range, ad esempio l'estremo dovrebbe essere incluso ma non lo è, o viceversa. Dal momento in cui la maggior parte degli errori si commette su tali test, si utilizza il testing dei valori limite.

13.10 Testing dei valori limite (Boundary values)

Tecnica che non sostituisce ma che va a complementare con la tecnica delle classi di equivalenza quando alcuni dei domini sono degli intervalli, per cui si partiziona i domini in sub-range. La tecnica della boundary value analysis vuole andare a verificare dove è commesso l'errore. Voglio capire quale sia il test fallito. Assumo quindi che la failure sia dovuta da uno solo dei test sui range, per cui voglio capire quali di questi sia. Si assume inoltre che se uno non è corretto, si ha la failure, per cui un solo valore non corretto mi basta per trovare la failure. Se questa è l'assunzione basta variare i valori minimi e massimi di 1, lasciando gli altri al valore nominale, ovvero quello centrale. Il calcolo dei casi di test necessari è $4n+1$, dove n è il numero di variabili. Si parte quindi da equivalence class testing, che in combinazione con boundary value analysis mi permette di scegliere più casi di test rispetto alla semplice equivalence class testing. La tecnica di boundary value analysis interviene nella fase di selezione di casi di test e non nella definizione delle combinazioni, cosa che invece equivalence class testing.

Questa tecnica funziona bene solo su range, su quantità fisiche limitate, potendo però fare test di robustezza. Si fa testing di robustezza prendendo anche un valore maggiore del

massimo ed uno minore del minimo.

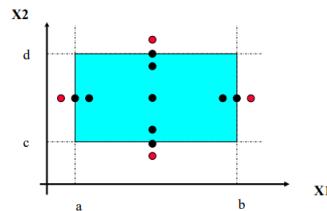
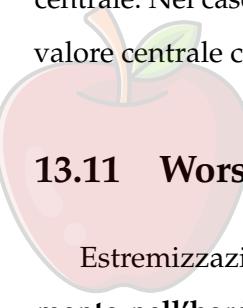


Figura 13.1: Boundary analysis

Esempio: Supponendo due classi di equivalenza. Il valore x_1 è $a \leq x_1 \leq b$, mentre $c \leq x_2 \leq d$. Con le classi di equivalenza si va a prendere il valore centrale di ogni classe di equivalenza. Non si riescono a testare gli estremi. Quello che si fa è andare a selezionare casi di test che si avvicinano agli estremi. Per ogni classe di equivalenza si vanno a segnare quindi 5 valori e non uno solo. I 5 valori sono: estremo superiore ed inferiore, valore immediatamente sopra l'estremo inferiore, un valore immediatamente sotto l'estremo superiore e un valore centrale. Nel caso in cui si abbiano 2 variabili saranno quindi necessari 9 casi di test, poiché il valore centrale corrisponde



13.11 Worst case testing

Estremizzazione della boundary value analysis. Essa è una tecnica che si usa maggiormente nell'hardware poichè assume che ci possano essere delle failures che vengono generate non quando fallisce uno dei test, ma quando più valori falliscono, ovvero vanno fuori range. Bisogna quindi testare tutte le possibilità di combinazioni in cui non va al limite una sola variabile ma più variabili. Se si hanno più possibilità per ogni variabile, devo andare a testare tutte le possibili combinazioni. È possibile applicare test di robustezza anche alla WCT

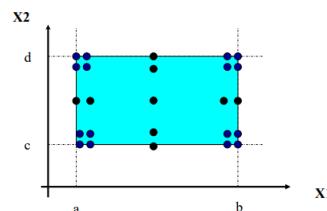


Figura 13.2: WCT

13.12 Metodi per ridurre i casi di prova

Il WECT esiste come criterio per opporsi come caso limite del minor numero di casi di test. il problema è che WECT si basa sul fatto che non ci sia nessuna interazione tra i casi di test. Il problema di SECT è che assume che gli errori possano derivare dalle interazioni delle classi di equivalenza, per cui si va a testare tutto. SECT assume anche che i valori non validi non dipendano dalle interazioni di classi di equivalenza, cosa che non è vera. L'obiettivo è ridurre il numero di combinazioni considerando solo i casi più significativi. Potrebbe capitare che il valore di una singola variabile mi determini l'esito dei test, ma tale valore con SECT sarebbe comunque combinata con tutti gli altri valori. è quindi necessario un metodo di selezione che ci permetta o di eseguire quella classe una sola volta o che ci permetta di scegliere con quali classi testarla. Alcune tecniche sono: **category partition**, tabella delle decisioni, grafo causa effetto e pairwise combinational testing che considera tutti i casi intermedi che vanno da WECT a SECT.

13.13 Category partition

Individua le funzioni che devono essere testate, potendo fare ciò per funzionalità interne ed esterne. Per il test di sistema, ad esempio, ogni caso d'uso è individuabile singolarmente. Per ognuna di queste funzionalità, si vanno ad individuare i parametri. Se è una funzione i parametri li trovo nell'interfaccia della funzione, se è un caso d'uso li trovo nella specifica. Oltre ai parametri, si vanno ad individuare gli oggetti dell'ambiente, ovvero gli oggetti con cui la nostra funzione interagisce e che, insieme ai valori dei parametri di input, determinano il risultato. Con equivalence class testing, trovavamo questi valori, individuavamo il dominio dei parametri e partizionavamo il dominio in classi di equivalenza. Il limite di tale tecnica è che funziona solo su range o enumerazioni, per cui il dominio dei valori è scalare e non strutturale.

Con **category partition**, per individuare il sottoinsieme delle stringhe che hanno un certo comportamento e quelle che hanno un comportamento diverso, lo si fa sulla base delle caratteristiche dei parametri. Molto spesso tali caratteristiche legano parametri tra loro, per cui ci dicono in che modo prendere dei parametri. Le caratteristiche sono le categorie che sono espresse su un dominio scalare. Le categorie sono ulteriormente suddivise. Queste classi di equivalenza in cui si divide il dominio delle categorie, sono chiamate, nel metodo del category partition, scelte. Si prendono valori normali, valori speciali e

valori limite. Il modo in cui andiamo a fare scelte usa quindi anche il boundary value analysis. I vantaggi sono che andiamo a considerare gli oggetti dell'ambiente, cosa che con l'equivalence class testing non veniva fatta. Un altro vantaggio è che con le categorie è applicabile a parametri non scalari ma strutturali. Come ultimo è possibile individuare vincoli, in modo che l'occorrenza di una scelta può influenzare l'esistenza di un'altra scelta. Tali vincoli sono espressi mediante proprietà e selettori

13.13.1 Vincoli

Se io voglio che una scelta B1 della categoria B possa essere combinata con una scelta A1 e non anche con la scelta A2, è necessario impostare dei vincoli. Si definisce una proprietà su A1 e per B1 si aggiunge un selettore. È necessario avere un selettore che faccia riferimento ad una property, altrimenti non ha senso. Grazie ai selettori posso decidere quali scelte sono valide e quali posso combinare, potendo generare test frames. I test frame sono le combinazioni di scelte valide, così come possono essere le combinazioni di classi di equivalenza. Una volta selezionati i test frames posso selezionare i test data.

Fin quando proprietà e selettori servono per scartare scelte incompatibili, allora si ha un partizionamento perfetto dello spazio di input, o almeno sull'input in cui è verificata la precondizione. Nel momento in cui proprietà e selettori li andiamo ad usare per ridurre casi di test per testare meno casi, allora stiamo facendo un quasi partizionamento.

Categoria A
SceltaA1 [property X, Z]
SceltaA2 [property Y, Z]
Categoria B
SceltaB1
SceltaB2 [if X and Z]

Figura 13.3: Vincoli

Le categorie sono definite sulla base di una specifica, non troppo sulla base delle precondizioni ma maggiormente sulla base delle precondizioni. Dato un array ad esempio, informazioni possono essere: numero di elementi, massimo e minimo nell'array, quale è la loro posizione nell'array... Tutti essi sono elementi che vanno a individuare una particolare configurazione dell'array.

Con category partition, per partizionare lo spazio di input, ovvero un array considerando il caso precedente, è necessario basarsi sulle diverse configurazioni che può assumere l'array. Ogni sottoinsieme rappresenta quindi un insieme di configurazioni di un'array che è simile per caratteristiche, dove le caratteristiche sono le categorie. Per individua-

re il partizionamento dello spazio input, è necessario il partizionamento di scelte delle categorie.

13.13.2 Proprietà delle scelte

è inoltre possibile fornire un'etichettatura, come **ERROR**, una proprietà che do ad una scelta nel caso in cui quella scelta non sia valida, non nel caso in cui la scelta mi determina un risultato. Usando una classe non valida andiamo a combinarla con scelte valide. Alcune proprietà sono:

- **ERROR:** può andare a interrompere l'esecuzione del programma senza inserire il valore degli altri parametri. La scelta **ERROR** può determinare anche un risultato ma comunque la funzione richiede anche altri parametri. Dovrà quindi esserci una sola scelta **ERROR**, ma le altre scelte dovranno comunque essere presenti, ovvero si dovrà avere una sola classe non valida seguita da classi valide. Se io ho diverse scelte valide per un parametro, la scelta **ERROR** verrà combinata con una sola scelta delle scelte dell'altro parametro, non con entrambe. La classe non valida infatti non verrà testata in combinazione con tutte le altre classi valide. Non tutti gli **ERROR** devono essere combinati con classi valide, mentre la combinazione con classi valide viene fatta a volte per vedere se il programma riconosce la classe non valida, segnata con **ERROR**.
- **SINGLE:** la classe deve essere testata una sola volta. Se quella classe posso combinarla con altre classi, è indifferente con quale delle classi venga combinata, purchè venga combinata una sola volta.

Tali proprietà possono essere ottenute anche mediante selettori, ma non è la stessa cosa. Nel caso si mettano i selettori, come nel caso di **SINGLE**, si andrà scegliere la classe specifica con cui combinare la classe, nel caso di **SINGLE** invece, la scelta della classe con cui combinarla sarà casuale, in quanto una vale l'altra.

13.13.3 Steps del category partition

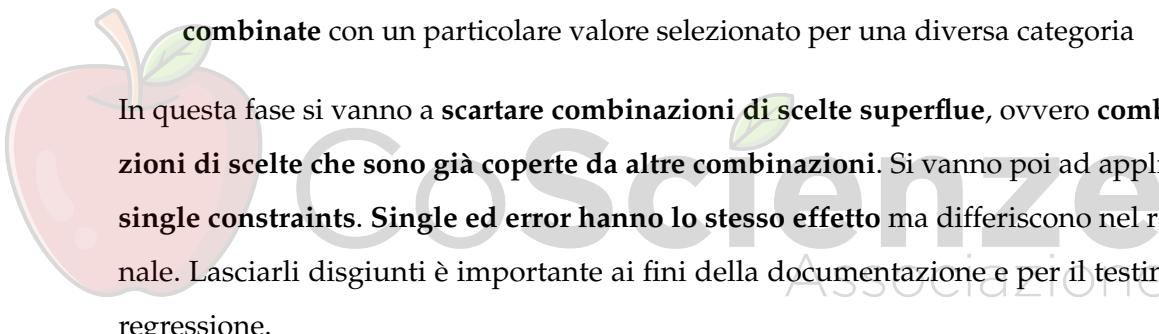
Gli steps da fare durante il category partition sono:

- **individuazione delle categorie:** Sulla base delle specifiche è necessario scegliere le categorie
- **individuazione delle scelte:** è necessario individuare le classi rappresentative di valori per ogni categoria. Si va ad ignorare le interazioni tra i valori di diverse

categorie. Le scelte possono essere individuate applicando **boundary value testing** (selezione valori estremi della classe, valori interni...) o **erroneous condition testing** (selezionare valori al di fuori del dominio)

- **introduzione dei vincoli:** scelte le combinazioni, non tutte sono possibili, **o comunque il numero di test cases è enorme.** Si introducono quindi dei **vincoli per eliminare combinazioni impossibili e ridurre le dimensioni della test suite** (se troppo grande). Si va quindi ad **applicare error constraints** per eliminare combinazioni impossibili, come **test che hanno un numero superiore ad uno di proprietà ERROR** ed infine si va ad applicare la **property constraints.** **Constraints come PROPERTY e IF-PROPERTY vanno ad eliminare combinazioni non valide** di valori (scelte):

- **PROPERTY:** raggruppa i valori di un singolo parametro per individuare sottoinsiemi di valori con proprietà comuni.
- **IF-PROPERTY:** limita le scelte di valori di una categoria che possono essere combinate con un particolare valore selezionato per una diversa categoria



In questa fase si vanno a scartare **combinazioni di scelte superflue**, ovvero **combinazioni di scelte che sono già coperte da altre combinazioni.** Si vanno poi ad applicare **single constraints.** **Single ed error hanno lo stesso effetto** ma differiscono nel razionale. Lasciarli disgiunti è importante ai fini della documentazione e per il testing di regressione.

13.13.4 Conclusioni

L'individuazione di parametri, condizioni di ambiente e categorie dipende fortemente dall'**esperienza del tester.** Rende le decisioni di testing esplicite ed aperte a revisioni. Una volta **completato il primo passo, la tecnica è semplice e può essere automatizzata.** Riducendo i casi di test, la tecnica è utile e rende **il testing sistematico più praticabile.**

Con il **category partition** l'idea è di individuare insiemi di input che hanno lo stesso **comportamento.** Ciò non può essere fatto andando a partizionare direttamente il dominio. A volte è necessario vedere l'interazione tra parametri del database, **avendo quindi delle caratteristiche da individuare.** è quindi necessario individuare i casi che mi rappresentano **insiemi di input che hanno un comportamento simile.** Si avrà quindi un partizionamento, un sottoinsieme, dello spazio di input. Ciò è che devono andare ad individuare, attraverso le categorie e poi le scelte. Le **categorie mi devono quindi aiutare a individuare delle scelte che**

mi partizionano in maniera adeguata lo spazio di input, ovvero individuare sottoinsiemi dello spazio input che hanno comportamento simile. Il category partition ci consente di progettare test. Si possono andare a **modificare vincoli, proprietà, selettori...** per ridurre o avere più casi di test. Scegliere che una particolare scelta che, seppur compatibile, non venga combinata con altre, è una decisione della progettazione del test.

13.14 Tabelle di decisioni

Consente di ridurre i casi di test, in quanto va ad **eliminare combinazioni incompatibili come per il category partition**. Essa però è meno sofisticata del category partition. Per il funzionamento, si indicano una serie di condizioni che rappresentano la combinazione di scelte ed a questa combinazione di condizioni corrisponde un'azione, ovvero la **risposta del sistema**. Si esprime quindi la postcondizione in tale modo, ovvero le condizioni che mi danno una certa risposta. Tali condizioni devono afferire alle variabili in input. La tabella è quindi fatta in 2 parti:

- **Sezione delle condizioni:** condizioni sulle variabili in input. Essa esprime relazioni tra variabili di decisione
- **Azioni:** risposte che devono essere prodotte quando una corrispondente combinazione di condizioni si verifica. Le azioni sono **indipendenti dall'ordine di input**, funzionando bene a livello di test di unità

Condizione	c1	True	True	False	True	False	False
c2	T				T		
c3	F			—	F		
Azione	a1	X	X		X		
a2	X				X	X	
a3		X		X	X		
a4			X	X	X		X

Figura 13.4: Tabella di decisioni

Inserendo un dato errato che dà un errore, per cui **non si possono fare altri inserimenti**, il **category partition riesce a gestire la cosa**, mentre con la tabella di decisioni che è indipendente dall'ordine di input, tale problema non sarà gestito. Si può avere la stessa risposta in corrispondenza di più combinazioni di condizioni e la stessa azione può trovarsi in corrispondenza di più combinazioni di condizioni.

c1: a, b, c triangle?		N						
c2: a = b?			Y					
c3: a = c?				Y	N			
c4: b = c?					Y	N		
a1: not a triangle	X							
a2: Scalene								
a3: Isosceles		X		X				
a4: Equilateral			X		X			
a5: Impossible					X			

Figura 13.5: Tabella di decisioni: Esempio

13.14.1 Condizioni di uso ideali

Il problema dell'**equivalence class testing** è che ogni input dovrebbe essere indipendente dall'altro, cosa differente in questo caso. è la **condizione che deve essere espressa sull'input a dire come scegliere l'input**. Molti problemi sono espressi in questo modo. Nella postcondizione i vari caso espressi da condizioni che valgono sugli input, per cui gli **input che soddisfano una condizione hanno lo stesso comportamento, ovvero la stessa azione**. Il concetto è sempre quello di partizionare lo spazio di input. Il problema è che la **tecnica delle classi di equivalenza è impraticabile come tecnica a meno che i valori siano indipendenti, e ciò spesso non capita, per cui la tabella di decisioni permette di dividere lo spazio input in modo più appropriato**. Tale tecnica va quindi usata quando è necessario **scegliere una risposta tra molte distinte**, in accordo a casi distinti delle variabili in input. Questi casi possono essere **modellati da espressioni booleane mutualmente esclusive sulle variabili di input**. La **risposta che deve essere prodotta non dipende dall'ordine in cui le variabili di input sono definite o valutate e non deve dipendere da input o output precedenti**.

13.14.2 Scalabilità

' Per n condizioni possono esserci al più 2^n **combinazioni di condizioni**. Di solito però ci sono meno, poichè sono **presenti dei valori "don't care"**, ovvero il **valore di verità di una delle condizioni non influenza il risultato**.

13.14.3 Condizioni speciali

I "don't care" possono corrispondere a casi differenti:

- **input sono necessari ma non hanno effetto**, ovvero il risultato non dipende da quel valore
- **input che possono essere omessi**: essi non hanno comunque effetto sul risultato
- **casi mutualmente esclusivi**

Altri casi speciali sono "non può succedere" poichè riflette qualche assunzione in cui alcuni input sono mutualmente esclusivi o non possono essere prodotti nell'ambiente. In molti casi, si indica con "non può succedere" anche quando le condizioni non sono veramente impossibili, ma si assume che un determinato caso non possa verificarsi. Ciò potrebbe essere causato da un'analisi errata, per cui condizioni si verificano anche se non dovrebbero. Errori di specifica potrebbero infatti portare ad errori nella progettazione del test. La condizione "non so" riflette un modello incompleto, ad esempio dovuto a documentazione incompleta. La maggior parte delle volte sono errori di specifica.

13.15 Metodi dei grafi causa effetto

Se le condizioni sono esplicite, ovvero la postcondizione è esplicita, definendo quindi cosa deve succedere purché essa valga è facile trovare le condizioni. Spesso però non è così facile trovare le condizioni, in quanto la specifica è formale e non informale, non essendo espressa in maniera così chiara.

Esiste quindi una tecnica chiamata dei grafi causa-effetto che aiuta a derivare tabelle di decisione. Individua le cause, che sono le condizioni sugli input, e gli effetti, ovvero gli output del sistema. Le cause vengono definite in termini di condizioni, in modo che sia vera o che sia falsa. La tecnica specifica esplicitamente dei vincoli su cause ed effetti, permettendo inoltre di selezionare combinazioni significative di cause-effetto che mi portano ad un risultato.

13.15.1 Struttura del grafo causa effetto

Ad ogni nodo corrisponde o una causa o un effetto, esprimibile quindi come grafo bipartito. Quello che cerchiamo di fare è interconnettere il grafo, capire quali sono le cause che producono quali effetti. Una linea tra causa ed effetto indica che la causa è una condizione necessaria dell'effetto, ma è possibile che una singola causa sia necessaria per più effetti ma anche che un singolo effetto possa avere più cause. Se un effetto ha due o più cause la relazione logica tra le cause è espressa inserendo AND se devono valere entrambe, o OR se almeno una è necessaria. Se ho una causa, la cui negazione è necessaria ad un effetto, si ha la linea tra i due nodi etichettata con NOT. Ciò potrebbe rendere complesso il grafo, per cui si usano nodi intermedi che raggruppano le combinazioni di cause per andare verso gli effetti.

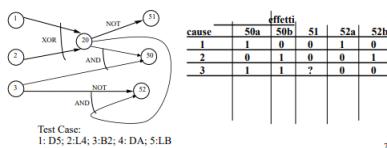


Figura 13.6: Grafo cause effetto

13.15.2 Derivazione dei casi di test

Producendo una tabella di decisione con:

- cause sulle righe
- effetti sulle colonne

Per ogni effetto, devo vedere quali siano le combinazioni di cause che produce quell'effetto.

Si va a ritroso sul grafo fino ad arrivare alle cause il cui valore di verità mi produce quell'effetto. Ad ogni colonna della tabella corrisponde un test case, che sarà la combinazione delle cause



13.16 Pairwise or n-way combinatorial testing

Con l'equivalence class testing abbiamo un problema se applichiamo WECT, non testando nella pratica nessuna combinazione di classi di equivalenza, poiché testiamo classi come se fossero indipendenti l'una dall'altra, pur non essendolo. SECT invece introduce molti casi di test, troppi. Una via di mezzo, in casi in cui non è possibile usare category partition ad esempio in caso di dati scalari o in assenza di vincoli, è possibile usare la pairwise combinatorial testing. Essa combina valori sistematicamente ma non esaustivamente. Il razionale è che molte interazioni avvengono solo tra due o pochi parametri o caratteristiche.

13.16.1 Pairwise combination (invece che esaustiva)

Genera combinazioni che efficientemente ricoprono tutte le coppie, triple o quadruple di classi di equivalenza, di scelte. Il razionale è che la maggior parte delle failure sono causate da valori singoli o combinazioni di pochi valori. Coprendo coppie si riduce il numero di casi di test ma consente di rilevare la maggior parte dei difetti.

CAPITOLO 14

Testing strutturale o testing white box

14.1 Introduzione: testing white box o testing strutturale

Applicare testing white box significa **basare i test sulla conoscenza della struttura del software, in particolare sul codice**. Si usano **flussi di controllo e strutture di controllo**. Tecnica non scalabile, quindi **usabile a livello di unità**. Essa è completa al testing funzionale. Pensare di andare a **selezionare casi di test in maniera completamente white box** è un'utopia. È molto più facile dare un significato ai test selezionati usando specifiche che non ai test usando il codice, stesso discorso vale per l'oracolo. È quindi **necessario usare tecniche white box quando il test funzionale non è sufficiente a garantire la copertura desiderata**. Con il testing strutturale per sé **non è possibile rilevare difetti che dipendono dalla mancata implementazione della specifica**. In generale i test white box si fondano sull'**adozione di criteri di copertura**. Per copertura si intende che si va a definire un insieme di casi di test tale che **gli oggetti di una definita classe siano attivati almeno una volta nell'esecuzione di casi di test**. È possibile definire la metrica di copertura: **test effectiveness ratio = numero oggetti coperti / numero oggetti totali**

Non è sempre possibile avere una copertura pari al 100% né tantomeno si riesce a decidere se è possibile avere una copertura del 100%. Ciò è legato a problemi di indecidibilità.

14.2 Selezionare i casi di test con tecniche white box

Scelto il criterio di copertura, è necessario selezionare i casi di test. Ogni test corrisponde all'esecuzione di un particolare cammino. Volendo selezionare casi di test che ci danno copertura bisogna individuare quali siano i cammini che ci danno copertura, ovvero coloro che coprono gli statement. Individuati tali cammini, posso fare esecuzione simbolica e trovare la condizione sulle variabili di input, condizione che soddisfa la path condition, che se vera consente di eseguire quel cammino. Agire in questo modo è molto costoso, per cui si sceglie, insieme ad un criterio di copertura white box, un criterio di copertura black box, funzionale.

Si individuano i casi di prova in accordo al criterio di copertura funzionale e lo si esegue, andando a misurare la copertura strutturale. Se abbiamo raggiunto la soglia, ovvero la test effectiveness ratio, ci fermiamo, altrimenti è necessario trovare altri casi di test che ci consentono di raggiungere la copertura desiderata, ovviamente applicando tecniche white box e vedendo quali sono le parti di codice che non sono state coperte, trovando un cammino che esegue quelle parti di codice e quindi trovare un input che esegue quel cammino, facendo esecuzione simbolica.

14.3 Statement coverage

Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i nodi del CFG, ovvero l'esecuzione di tutte le istruzioni di input. Statement coverage e node coverage sono quindi la stessa cosa. Per eseguire tutti gli statement potrebbe bastare un solo cammino, in base al tipo di problema. TER = numero statement eseguiti / numero statement totali

14.4 Copertura delle decisioni o branch coverage

Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i rami del CFG, ovvero l'esecuzione di tutte le decisioni. Mi basta che per ogni decisione io eseguo il ramo vero ed il ramo falso. Se avviene questo, per ogni decisione sono necessari due casi di test, uno che fa diventare vera la condizione e una che la fa diventare falsa. Se avrò questo, riuscirò a coprire tutti i branch, tutti gli archi del CFG. Se copro tutti i branch compro tutti i nodi, per cui branch coverage implica statement coverage. Non è garantito trovare la failure poiché le decisioni sono composte da più condizioni in

AND o OR e in alcuni casi branch coverage non è sufficiente. Possono esserci più modi diversi per coprire le decisioni. Esso infatti permette un **quasi-perfezionamento**. TER = numero branch eseguiti / numero branch totali

14.5 Copertura delle condizioni

Dato un programma P, viene definito un **insieme di test case la cui esecuzione implica l'esecuzione di tutte le condizioni** (valori vero e falso delle componenti relazionali dei predicati) caratterizzanti le decisioni di P. Significa che per ogni condizione vogliamo che ci siano almeno due casi di test, una che la fa diventare vera ed una falsa. Non si coprono le **decisioni**, ovvero il predicato della struttura di controllo, ma le singole condizioni che si trovano all'interno del predicato. La copertura delle condizioni non implica la copertura delle decisioni

14.6 Copertura di decisioni e condizioni

Dato un programma P, viene definito un insieme di **test case la cui esecuzione implica l'esecuzione di tutti i predicati** (definito anche come decisioni) e di tutte le condizioni caratterizzanti i predicati in P. Si hanno quindi due tipi:

- **multiple condition coverage**: per ogni predicato vengono considerate tutte le combinazioni di condizioni. Il numero di condizioni è 2^n , dove n è il numero di condizioni
- **modified condition decision coverage (MCDC)**: per ogni predicato vengono considerate solo le combinazioni di valori delle condizioni per le quali una delle condizioni determina il valore di verità della decisione. In pratica si costruisce la tabella di verità delle condizioni. In questa tabella è sempre possibile trovare due coppie di combinazioni in cui il valore di verità del risultato dipende solo da una condizione. La condizione cambia mentre le altre no. Significa quindi che il valore di verità di quella condizione determina il valore di verità della decisione, che è indipendente dalle altre condizioni. Per ogni clausola si prende quindi il predicato che la rende vera e che la rende falsa e che cambiando va a cambiare il valore di verità della decisione. Per ogni condizione si vanno quindi a trovare casi di test che fanno diventare una volta vera ed una volta falsa la condizione. Si vanno quindi a prendere i casi di test che non solo fanno diventare vera e falsa la condizione, ma che fanno diventare vero e falso anche il predicato, non cambiando le altre condizioni. Ciò viene fatto per

ogni condizione. Riuscirò sempre a trovare una **coppia** in cui il valore di verità della **condizione determina il valore di verità della decisione**. Il numero di test case è 2^n , dove n è il numero di condizioni

14.7 Copertura dei cammini o path coverage

Dato un programma P viene definito un **insieme di test case** la cui esecuzione implica l'**attraversamento di tutti i cammini GFC(P)**. Eseguire lo stesso percorso in cui è presente un **ciclo è complesso**, in quanto se varia il **numero di volte** che si esegue il ciclo, cambierà anche **il percorso**, considerato differente rispetto alle volte precedenti. Alcuni problemi potrebbero essere legati al **numero infinito o alto di cammini o agli infeasible path**, percorsi per cui non esiste un input che sia in grado di eseguirli. **Un numero di cammini infinito implica la presenza di circuiti**. Ci sono alcune tecniche per limitare il numero di cammini. Essa è l'**unica tecnica a darmi un partizionamento perfetto**. TER = **numero cammini eseguiti / numero cammini totali**

Se si hanno dei cicli si potrebbe avere un numero di cammini potenzialmente infinito. Per limitare il numero di cammini è necessario applicare alcune tecniche, come **criterio di n-copertura dei cicli**, metodi basati su **exemplar-paths**, metodi dei **cammini linearmente indipendenti** (Mc Cabe) e **metodi basati su data-flow**.

14.8 Criterio di n-copertura dei cicli

Si seleziona un insieme di test case che **garantisce l'esecuzione dei cammini contenenti un numero di iterazioni di ogni ciclo non superiore ad n**. Ogni ciclo **deve essere quindi eseguito da 0 ad n volte**. Di solito $n=2$. Il ciclo viene quindi eseguito 0, 1 e 2 volte. Il criterio di **1-copertura**, ovvero con $n=1$, **implica il criterio di copertura del branch**, nel caso in cui **una volta entro nel ciclo ed un'altra non ci entro**. Tale criterio è quindi un **qualcosa in più** rispetto al criterio di **branch coverage**. Se si decide di eseguire il ciclo una sola volta, in cui prima il ciclo è **vero** e poi è **falso**, il caso di test è lo stesso per cui il branch non è coperto. Per coprire il branch bisogna **non entrare proprio una volta e poi entrare un'altra**. Avendo dei vettori, **su sistemi complessi, avere n=2 non basta**, in quanto le relazioni potrebbero essere non in cicli consecutivi ma relazioni con i iterazioni dopo.

14.9 Metodo degli exemplar path

L'insieme dei cammini del grafo di controllo viene partizionato in un numero finito di classi di equivalenza. Per il criterio di copertura bisogna trovare l'insieme di casi di test che assicuri l'attraversamento almeno una volta di almeno un cammino per ogni classe. Trovo quindi gli exemplar path, ovvero i rappresentati delle varie classi. Due cammini sono assegnati alla stessa classe se essi differiscono univocamente nel numero di volte per il quale un ciclo sul cammino viene percorso.

14.10 Partizionamento perfetto e quasi-perfetto

Si riesce ad ottenere un partizionamento perfetto dello spazio di input mediante il **path coverage**, poichè i casi di test che coprono un cammino non possono coprire un altro cammino. In generale, per le altre coperture, l'insieme di cammini che copre lo statement si sovrappone all'insieme di cammini che copre altri statement. Un criterio che mi chiede più casi di test è quello che partiziona di più, mentre il criterio che mi chiede meno caso di test è quello che partiziona meno lo spazio di input, in quanto va a prendere casi di test che mi coprono più elementi contemporaneamente.

14.11 Esecuzione simbolica

Bisogna eseguire un programma. Non lo si esegue usando valori reali ma usando valori simbolici. L'esecuzione procede come esecuzione normale ma non sono elaborati i valori, bensì formule formate da valori simbolici sull'input. Ogni variabile intermedia sarà un'espressione sui valori simbolici delle variabili in input. Anche gli output saranno delle formule. Finchè si ha una sequenza, si sostituiscono espressioni simboliche al posto di variabili, andando a semplificare. Avendo un'istruzione condizionale, anche il predicato sarà espresso in funzione dei valori simbolici delle variabili in input, poichè anche lì si andranno a sostituire espressioni simboliche delle variabili che compaiono nell'espressione condizionale. Si ottiene quindi un'espressione booleana sui valori simbolici delle variabili in input. Il valore dei predicati potrà assumere valore vero o falso, in dipendenza dei valori che assumono i valori simbolici di input. Si potrebbero trovare degli input che rendono il predicato vero ed altri che rendono il predicato falso. Se un predicato è sempre vero vuol dire che ciò che si troverà dal lato falso è unfeasible. Se si hanno dei cicli, ogni volta che

viene eseguito, cambia il valore simbolico. Nel ciclo, il numero di giri dipende da cosa voglio testare e da quante volte voglio testare il ciclo.

14.11.1 Path condition

Lo stato delle variabili a seconda del percorso che ho eseguito sarà differente. Il risultato alla fine, lo stato delle variabili in output, sarà diverso sui diversi cammini di esecuzione. Per definire in quale caso su quale cammino c'è stato un risultato mentre su un altro cammino c'è stato l'altro risultato è necessario capire quale sia l'input o cammino, piuttosto che l'espressione, che mi ha portato a quel risultato. La path condition definisce la condizione che deve essere verificata affinché quel particolare cammino di esecuzione venga eseguito. Anche la path condition è espressa come una condizione booleana sui valori simbolici delle variabili in input.

La path condition è inizialmente vera, in quanto le prime istruzioni vengono eseguite sempre. Ogni volta che però si incontra una condizione, devo calcolare l'espressione simbolica di quella condizione e devo dire in quali casi eseguo il ramo vero ed in quali casi eseguo il ramo falso. Una volta scelti i cammini di esecuzione, per selezionare i casi di test devo trovare degli input che soddisfano le path condition dei cammini che ho scelto. Farlo però è un problema NP. In SAT i valori erano booleani, in questo caso si hanno condizioni.

14.11.2 Feasible & unfeasible path

Si ha un feasible path quando si ha un insieme di dati in input che soddisfa la path condition. Un path è unfeasible se non esiste un insieme di dati in input, un test case, che soddisfa la path condition. Dimostrare che due espressioni simboliche sono equivalenti, ovvero che due cammini del programma calcolano la stessa funzione, è indecidibile.

14.12 Execution three

L'execution tree cattura le diverse decisioni (come istruzioni "if" e "else") e le possibili strade che il programma può percorrere in base alle diverse condizioni e allo stato delle variabili durante l'esecuzione. Questo tipo di rappresentazione può essere utile per analizzare la complessità del flusso di controllo del programma, identificare possibili cammini critici o particolari scenari di esecuzione, e per valutare la copertura dei test.

14.13 Problemi di raggiungibilità indecidibili

La path condition è un AND di condizioni e quindi equivale ad un sistema di disequazioni. Decidere se esiste un insieme di input che soddisfa la path condition è un problema indecidibile. Dire quindi se esiste almeno un caso di test che esegue un cammino di esecuzione è un problema indecidibile, per cui dire se un path è unfeasible è indecidibile. Dire inoltre che se esiste un input che esegue un'istruzione o un particolare ramo, allora è indecidibile.

14.14 Metodo di McCabe

McCabe ha identificato quale sia il numero massimo di casi di test che mi servono per ottenere la copertura di un intero sistema. Essa è una tecnica basata sui cammini, in quanto quello che si fa è individuare l'insieme di cammini che garantisce la copertura dei branch. Si ha quindi un modo operativo per trovare i cammini che mi garantiscono la copertura dei branch. McCabe ha identificato il numero massimo di casi di test necessari per la copertura del branch, definendo un upper bound al numero di cammini necessari per il branch coverage.

Tale tecnica si chiama il metodo dei cammini linearmente indipendenti. Si avranno dei vettori, per cui possiamo dire se essi siano linearmente indipendenti o meno, ovvero se nessuno di essi può essere ottenuto come combinazione lineare degli altri. Moltiplicato per un numero ciascuno dei vettori, ottengo dei valori la cui somma mi da uno dei vettori.

14.14.1 Funzionamento del metodo di McCabe

Si usa un CFG(P), dove P è il programma. Siano n_N ed n_E il numero di nodi e archi di un grafo del flusso di controllo G, un cammino può essere rappresentato come un vettore di n_E elementi, ogni elemento rappresenta il numero di occorrenze dell'arco sul cammino. Tale rappresentazione è afferente ai cammini non agli archi. Ogni vettore rappresenterà un cammino. Per avere un cammino rappresentato da un vettore che però sia confrontabile con gli altri, è necessario che tutti i cammini abbiano gli stessi elementi, ma tutti i cammini non hanno gli stessi archi, per cui la soluzione è rappresentare ogni cammino all'interno del grafo come un vettore di tutti gli archi all'interno del grafo stesso. Per ogni arco, rispetto al cammino, sarà necessario definire se esso appartiene o meno al cammino. Parlando però di dipendenza lineare, un vettore booleano non è sufficiente. Non bisogna solo dire se l'arco

esiste all'interno del cammino, ma è necessario dire anche quante volte un arco è percorso all'interno di quel cammino.

Si prende il CFG(P) e si rappresentano i cammini con un vettore di tutti gli archi presenti nel CFG. **Per ogni cammino, il valore dell'arco sarà 0 se l'arco non è presente**, mentre **k se è presente**, dove k indica il numero di volte che quell'arco è percorso all'interno del cammino. In tal modo **possiamo osservare se dei cammini sono linearmente indipendenti**. **Se un cammino attraversa almeno un arco non ancora percorso, allora quel cammino è linearmente indipendente rispetto agli altri**, questo perché **negli altri cammini, il valore per quell'arco sarà 0**, mentre il valore di quell'arco sul cammino che sto considerando sarà diverso da 0. Non si potrà quindi ottenere mediante combinazione lineare di 0, un valore diverso da 0. Essa è una **condizione sufficiente ma non necessaria**. Non è necessaria perché potrei avere che un cammino percorre archi già percorsi ma è comunque linearmente indipendente.

14.14.2 Insieme dei cammini base

Il metodo di McCabe implica il criterio di copertura del branch. Se prendo un insieme massimo di cammini linearmente indipendenti, per ogni altro cammino che vado ad aggiungere, questo cammino non può coprire archi non coperti da altri, perché se fosse così sarebbe ancora linearmente indipendente. Dato un insieme massimo di cammini linearmente indipendenti, questo insieme copre i branch, poiché se non fosse coperto potrei prendere un cammino che copre quel branch e aggiungerlo, in quanto linearmente indipendente, per cui l'insieme di partenza non sarebbe l'insieme massimo.

14.14.3 Criterio di McCabe

L'insieme massimale di cammini linearmente indipendenti è detto insieme dei cammini base. Tutti gli altri cammini all'interno del CFG(P) possono generati da una combinazione lineare dei cammini di base. Dato un programma l'insieme dei cammini di base non è unico. Il criterio di copertura di McCabe è quello di trovare un insieme di casi di test che garantisce l'esecuzione almeno una volta di ogni cammino all'interno dell'insieme di cammini di base. La copertura di McCabe implica la copertura dei branch, non vale il contrario, per cui il criterio di McCabe è più forte.

14.14.4 Teoria ciclomatica dei grafi e McCabe

McCabe ha applicato la **teoria ciclomatica dei grafi, applicabile a grafi fortemente connessi**. Il fatto che sia **fortemente connesso** implica la presenza di cicli. Il CFG(P) non è di base un **nodo fortemente connesso**, perchè ogni nodo è raggiungibile dal nodo iniziale e ogni nodo può raggiungere il finale. McCabe ha aggiunto un arco fittizio dall'iniziale al finale, ottenendo un **grafo fortemente connesso**, per cui da ogni nodo è possibile raggiungere ogni altro nodo. Ha poi applicato al CFG la **teoria ciclomatica dei grafi**. Il numero di McCabe, ovvero il **numero massimo di cammini linearmente indipendenti**, è anche detto **numero ciclomatico o complessità ciclomatica di McCabe**. Ci sono vari modi per calcolare il numero ciclomatico di McCabe.

14.14.5 Calcolare il numero ciclomatico di McCabe

Il **numero ciclomatico** è pari al **numero di regioni chiuse all'interno del grafo**. Avendo un unico ciclo, ovvero nodi con un solo arco uscente, ho un'unica regione chiusa. Se aggiungo un arco, il numero di archi sarà 1 in più rispetto al numero di regioni chiuse. Nel CFG ogni nodo predicato, ovvero una decisione indicato come n_D , aggiunge una regione chiusa, per cui all'interno del grafo, la metrica di McCabe è calcolabile come:

$$\text{numero ciclomatico} = \text{numero regioni chiuse (nodi decisione)} + 1.$$

$$\text{numero ciclomatico} = \text{numero nodi predicato (nodi decisione)} + 1.$$

$$\text{numero ciclomatico} = n_E - n_N + 2.$$

In un **grafo ciclico**, se non si hanno decisioni, nella seconda formula va sommato 1 e non 2. Si ha in alternativa 2 come valore perchè è necessario calcolare l'arco fittizio che viene aggiunto. In un CFG, se non ci sono decisioni il **numero ciclomatico** è pari a 0 senza considerare l'arco fittizio, 1 altrimenti. Esso però va sempre considerato. Nel caso in cui però si stiano considerando le decisioni, l'ultimo arco non coinvolge le decisioni, per cui il calcolo rimane lo stesso. Se il **predicato** è innestato ogni percorso copre archi non coperti da altri. Se invece i **predicati** sono in serie ci sono percorsi che coprono nodi già coperti da altri.

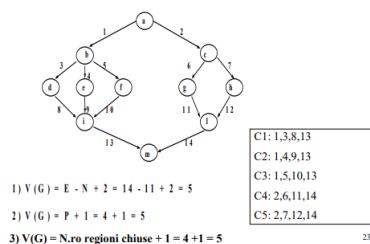


Figura 14.1: Numero ciclomatico di McCabe

Esempio: Supponiamo che un nodo abbia un solo successore per cui tutti i nodi si trovano sullo stesso cerchio. Il numero ciclomatico di McCabe sarà quindi 1. Se si aggiunge un nodo che ha due successori, il numero ciclomatico sarà 2. Se il nodo ha tre successori, il numero ciclomatico sarà 3...

14.14.6 Ricerca dei percorsi indipendenti

La tecnica per trovare i cammini linearmente indipendenti che mi dà la copertura dei branch è il **metodo dei cammini di base**, detta anche **tecnica delle baseline successive**. Ad ogni passo vado ad aggiungere un arco non coperto dagli altri. Modifico quindi la baseline ad ogni passo aggiungendo un nuovo cammino. È da tener conto inoltre i nodi già percorsi, memorizzando quale sia il branch percorso. Arrivando al nodo decisione, se ho percorso già un arco, vado a percorrere l'altro. Mi devo inoltre memorizzare il percorso che mi ha permesso di raggiungere uno specifico nodo. È possibile infatti passare per archi uscenti da decisioni che possono portare ad archi non esplorati precedentemente, in quanto si avevano decisioni in cascata.

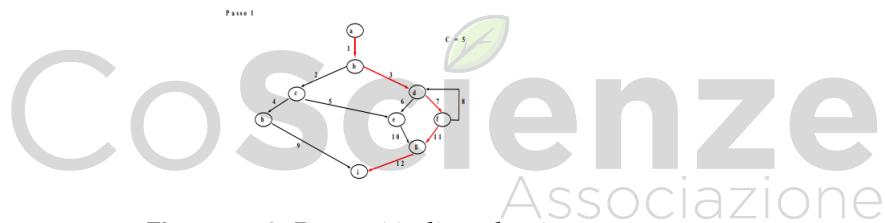


Figura 14.2: Percorsi indipendenti

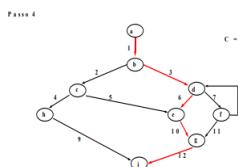


Figura 14.3: Percorsi indipendenti

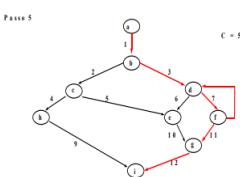


Figura 14.4: Percorsi indipendenti

Se ho **n if in serie**, il **numero di McCabe** è $n+1$, mentre il numero di **possibili cammini** è 2^n , mentre il **numero di cammini necessari per coprire i branch** è 2. Questa grande differenza

si ha solo nel caso di if in serie. Nel caso infatti in cui non si abbiano if in serie, il numero di cammini realmente indipendenti coincide con il numero di cammini e con il numero di cammini percorsi per coprire i branch. Il programma più testabile, tra programma con if in serie ed if in cascata, è quello con if in cascata.

14.15 Metodi fondati su data flow

Tecnica che va ad individuare dei cammini da testare, non basandosi però sul concetto di branch come McCabe, ma va a considerare i def use path, ovvero classi di cammini definizione-uso. Dato un programma P, sia CFG(P) il suo grafo del flusso di controllo. Un cammino "x, n1, n2, ..., nk, y" dal nodo x al nodo y è un definition clear path, rispetto ad una variabile v, se non ci sono definizioni di v su nessuno dei nodi ni (possono esserci definizioni di v sui nodi x e y). Un definition clear path "x, n1, n2, ..., nk, y" dal nodo x al nodo y è un definition use path rispetto ad una variabile v se c'è una definizione di v ad x ed un uso di v ad y.

14.15.1 Tipi delle variabili

Le tecniche basate su data-flow distinguono due tipi di uso della variabile:

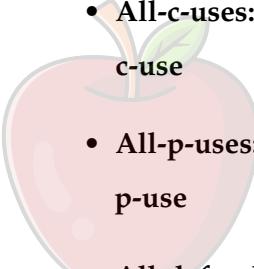
- Computational use: si ha quando una variabile v è usata in espressioni per il calcolo di un'altra variabile o in istruzioni di output
- predicate use: si ha quando la variabile v è usata in un predicato che condiziona il flusso di controllo

L'idea dei criteri di copertura è di coprire classi di du-path, piuttosto che coprire nodi o branch

14.16 Tipi di path

Esistono vari tipi di path:

- All-du-path: deve essere eseguito ogni du-path da ogni definizione ad ogni uso. Esso è molto vicino ad all-path
- All-uses: deve essere eseguito almeno un du-path da ogni definizione ad ogni uso. Se esistono da una definizione ad un uso più du-path, è necessario eseguirne almeno uno

- **All-p-uses/some-c-uses:** deve essere eseguito almeno un du-path da ogni definizione ad ogni p-use. Ciò significa che io faccio in modo che quel predicato assuma due valori diversi, uscendo sia sul ramo vero che sul ramo falso. Se ho una definizione che raggiunge più p-use, devo testare almeno un du-path che dalla definizione arriva a ciascuno degli usi predicativi. Testare mediante du-path un uso predicativo vuol dire testare due percorsi, quello con valore vero e quello con valore falso. Avrà quindi due casi di test, di cui uno esegue il ramo vero ed uno che esegue il ramo falso. Possono esserci più percorsi che vanno dalla definizione alla p-use. Se una definizione non raggiunge alcun uso predicativo, la variabile non viene utilizzata in nessun predicato, per cui è necessario raggiungere un uso computazionale
- **All-c-uses/some-p-uses:** deve essere eseguito almeno un du-path da ogni definizione ad ogni c-use, se una definizione non raggiunge nessun c-use allora va eseguito almeno un du-path dalla definizione alla p-use
-  **All-c-uses:** deve essere eseguito almeno un du-path da ogni definizione ad ogni suo c-use
- **All-p-uses:** deve essere eseguito almeno un du-path da ogni definizione ad ogni suo p-use
- **All-defs:** almeno un du-path da ogni definizione ad un uso (p-use o c-use). In questo caso si sta testando le definizioni

All-path è il più forte mentre all-defs è il più debole. Alcune tecniche come All-c-uses/some-p-uses e viceversa, non garantiscono la copertura del branch, per cui sono stati inseriti ma non vengono utilizzati. Stabilire se una definizione raggiunge un uso, è un problema indicibile, poiché equivale a dire se un cammino è feasible o unfeasible. Quando ci sono i cammini unfeasible ci sono anche dei du-path unfeasible, il che è un problema.

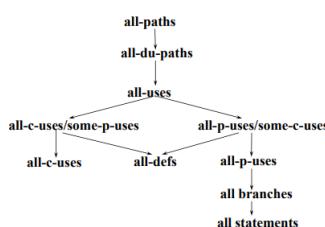


Figura 14.5: Tipi di path

14.17 Criteri strutturali e livelli di testing

La selezione di casi di test mediante criteri di copertura strutturali è principalmente indicata per il testing di unità. I criteri basati su cammini sono poco scalabili.

14.18 GCOV

gcov è uno strumento che consente di misurare la copertura strutturale, quindi branch e statements, di codice C. Per utilizzare gcov è necessario instrumentare il codice. Compilando un file devo usare degli switch che consentono di instrumentare il codice che poi darà infine le informazioni che ci servono. Devo quindi andare ad aggiungere del codice che mi produce delle informazioni sulla copertura dell'esecuzione. Tale codice viene aggiunto automaticamente all'atto della compilazione, andando ad inserire degli switch. Si userà poi "gcov nome file" che recupererà le informazioni aggiunte al codice mediante switch. Durante l'esecuzione, abbiamo aggiunto del codice che ha scritto le informazioni che ci servivano all'interno di un file .da, informazioni che verranno usate poi da gcov, facendo un matching tra file delle informazioni e file del programma. Con opzione "-b" si otterranno informazioni sulle branch. Il fatto che ci siano 2 branch al 100% indica che non ci sono stati né break né continue, per cui sia l'incremento che il controllo sono stati sempre eseguiti.

14.19 Software testing

Il testing di sistemi software OO deve tener conto di alcune caratteristiche: astrazione sui dati, ereditarietà, polimorfismo e binding dinamico. Molto spesso il software OO è generico, gestisce le eccezioni e si ha la concorrenza. Cambia il concetto di unità per cui si hanno dati ed operazioni. Il testing di integrazione è diverso dal testing di integrazione tradizionale. L'infrastruttura è diversa, come anche la generazione di casi di test. Anche la generazione di casi di test è diversa. È necessario testare il singolo metodo ma anche la classe

14.20 Stato ed information hiding

Per accedere direttamente allo stato privato della classe, piuttosto che utilizzare getter e setter dando per assunto che siano corretti, si potrebbe utilizzare la reflection, ovvero auto-ispezione. Per il testing dei metodi è possibile usare le tecniche tradizionali viste in

precedenza. Ciò viene definito intramethod in quanto si testa il singolo metodo e non la classe. **Testare la classe vuol dire testare il comportamento**, che è dato non dall'esecuzione del singolo metodo, **ma dal come una classe reagisce ad una serie di eventi**, serie di messaggi inviati alla classe. Posso quindi **usare degli scenari, una sequenza di invocazione di metodi che sono equivalenti**. Per sapere se si arriva nello stato corretto, si **usa una seconda sequenza di invocazioni di metodi alternativa, che dovrebbe però portarmi sullo stesso stato**, potendo **confrontare il risultato delle due sequenze equivalenti**. Mediante specifiche formali ciò sarebbe più semplice. In **alternativa sarà necessaria la conoscenza della specifica della classe per definire sequenze equivalenti e non equivalenti**. Ciò sarà utile per produrre l'oracolo.

14.21 Macchine a stati finiti e generazione di casi di test

L'idea è di **esprimere un comportamento della classe mediante una macchina a stati finiti**, macchina che descrive il modo in cui la **classe reagisce agli stimoli e cambia stato**. Si potrebbe considerare che **ogni percorso sulla macchina a stati finiti è un caso di test**. Posso usare approcci basati sulla specifica oppure si può fare reverse engineering dal codice, tirare fuori la **macchina a stati finiti e derivare casi di test**. I cammini di una macchina a stati finiti corrispondono a sequenze di esecuzione, per cui posso andare a selezionare tutti i **cammini senza cicli**. Esistono diverse tecniche per generare la macchina a stati finiti e per **limitare il numero di cammini** e questo dipende dai criteri di copertura e dalle tecniche di generazione di casi di test.

14.21.1 Macchina a stati gerarchica

Si possono avere **macchine a stati gerarchica**. In una macchina si può avere:

- **decomposizione in OR:** una **transizione in ingresso a uno stato decomposto in OR** può rappresentare più transizioni elementari che richiedono test distinti
- **decomposizione in AND:** uno **stato decomposto in AND rappresenta il prodotto cartesiano degli stati componenti**. Tutte le possibili coppie devono essere testate.

Essendoci della concorrenza è necessario forzare il comportamento del programma a stare in una combinazioni di stati. Se **concorrente**, si potrebbe infatti avere una combinazione qualunque in quanto dipende dallo scheduler e da altri fattori. Se voglio testare però, devo forzare lo scheduler a combinare gli stati in un modo specifico. Esso è un intervento pesante che **deve essere fatta sull'infrastruttura runtime del sistema**. Anche in questi casi si

può avere un **missmatch** tra specifica ed implementazione. Con l'integrazione si hanno problemi a livello di polimorfismo e binding dinamico.

14.21.2 Criterio di copertura

La macchina a stati finiti può essere utilizzata per **identificare criteri di copertura strutturale**:

- **generare sequenze di test a partire da specifiche**
- **generare macchina a stati finiti a partire da codice**
- **la qualità del test generato** (o del software) può essere misurata dal grado di copertura dei cammini della macchina a stati finiti.



CAPITOLO 15

Testing di integrazione

15.1 Testing di integrazione basato su branch

Si usano i sequence diagram per testare le varie partizioni. Mentre precedentemente si testa tutto il grafo, usando l'approccio basato su sequence diagram, vado a testare pezzi di integrazione in maniera verticale, in quanto ogni sequence diagram segue la funzionalità dal presentation layer fino ai dati. Testando i vari casi d'uso andando poi ad integrare, vado a testare le diverse partizioni in cui il sistema è decomposto. Se i sequence diagram descrivono un'unica interazione allora è facile eseguirli. Essi potrebbero però essere più articolati, avendo alternative e potendo generare più sequenze di esecuzione sullo stesso sequence diagram. In tal caso è necessario effettuare una selezione top-down.

15.2 Problemi del testing di integrazione

I problemi del testing di integrazione sono:

- ereditarietà
- polimorfismo

15.3 Ereditarietà

Essendoci l'ereditarietà alcune operazioni vengono mantenute così come sono, altre aggiunte e altre sovrascritte. Nel testing di integrazione si fa prima il testing della superclasse e poi della sottoclasse. Anche nel caso in cui un metodo della superclasse non cambi è necessario rieffettuare un test nella sottoclasse, questo poichè potrebbe contenere al suo interno chiamate a metodi che sono però stati modificati. I test sulla classe definiti per la superclasse non possono essere usati anche per la sottoclasse. Non tutti i metodi devono essere ritestati. è quindi necessario andare a capire, per ogni operazione, se è possibile riusare i casi di test che ho già definito per la superclasse. Può essere necessario verificare la compatibilità tra superclasse e sottoclasse. è quindi necessario identificare le proprietà da testare: operazioni aggiunte, operazioni ridefinite, operazioni invariate ma influenzate dal nuovo contesto. Può essere necessario verificare la compatibilità di comportamento tra metodi omonimi in una relazione classe-sottoclasse per definire se riusare i test o definirne di specifici.

15.3.1 Possibili soluzioni all'ereditarietà

Possibili soluzioni sono:

- **appiattimento:** si ritesta ogni metodo ignorandone la gerarchia di appartenenza.
Vantaggioso per il riuso dei test ma altamente ridondante e quindi inefficiente
- **test incrementale:** definire i tipi di una classe derivata, definire quindi gli scenari di test che in qualche modo vanno aggiunti riusando uno scenario precedente. è inoltre necessario identificare test da riusare... essa è più efficiente ma più costosa. è costosa la progettazione del test, ma è efficiente l'esecuzione.

15.4 Polimorfismo

Nei linguaggi procedurali classici, le chiamate a procedure sono associate staticamente al codice corrispondente. Nei linguaggi orientati a oggetti si avrà un riferimento che può denotare oggetti appartenenti a diverse classi in relazione classe-sottoclasse o tipo-sottotipo (polimorfismo), per cui il tipo statico ed il tipo dinamico possono essere diverse. Si possono avere più implementazioni della stessa operazione, per cui a runtime, in base al tipo dinamico, ovvero in base alla classe di appartenenza dell'oggetto, decido quale dei metodi

debba essere eseguito. Ciò però afferisce anche alle operazioni chiamate da un'operazione che viene ereditata.

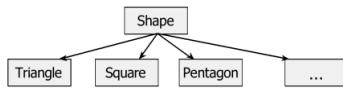


Figura 15.1: Polimorfismo

15.4.1 Possibili soluzioni del polimorfismo

Sapendo solo a runtime quale delle funzioni con stesso nome verrà chiamata, per la definizione di test case, una possibilità sarebbe quella di testare tutte le possibili classi della gerarchia. Non è però necessario testare tutto in quanto si potrebbe capire il tipo dinamico che l'oggetto può assumere mediante analisi statica. Arrivando alla chiamata che si sta considerando, è possibile vedere il tipo che assumerà il parametro attuale, sottotipo della classe generica. È necessario fare un'analisi di raggiungibilità, un reachable uses. Devo prendere tutti i punti in cui il metodo è chiamato e andare all'indietro per vedere qual'è il tipo di istanziazione della variabile che può raggiungere la chiamata a metodo che ci interessa. In tal caso potrebbero esserci degli unfeasible path che però non vengono evidenziati dall'analisi statica. È più semplice usare testing black box.

Con il testing white box (strutturale), invece, potrebbe diventare impraticabile poiché si ha un'esplosione combinatoria di possibili casi. L'idea sarebbe quella di utilizzare un criterio di copertura casuale o più accurato, basato quindi sull'analisi del codice. Per testare le classi generiche è necessario usare classi stub, come ad esempio String, in quanto classe predefinita e ha come proprietà quella di avere un tipo ordinato, per cui è utilizzabile anche per testare b-tree...

15.5 Gestione delle eccezioni

Nel punto in cui viene lanciata l'eccezione, c'è un branch senza test. Le eccezioni modificano il flusso di controllo senza la presenza di un branch esplicito è però necessario selezionare casi di test per capire se c'è o meno stata un'eccezione. I casi di test devono inoltre sollevare l'eccezione. Per scegliere tali casi di test è necessario usare gli indici di copertura:

- **copertura ottimale:** sollevare tutte le possibili eccezioni in tutti i punti del codice in cui è possibile farlo (può non essere praticabile). Ciò potrebbe essere **poco praticabile**, perchè richiederebbe molti casi di test
- **copertura minima:** sollevare almeno una volta ogni tipo di eccezione

15.6 Concorrenza

Il problema principale della concorrenza è il **non determinismo**. Avendo degli oggetti concorrenti, ognuno ha una sequenza di stati per cui potrò trovarmi in una qualunque combinazione di stati differenti. Lo stato in cui mi trovo non è quindi deterministico. Non mi basta soltanto avere input ed output dal momento in cui la failure non si verifica quando c'è una particolare combinazione di stati negli oggetti concorrenti e avendo solo input ed output dei vari oggetti, l'output potrebbe essere corretto solo perchè gli oggetti si sono trovati in una combinazione di stati che non mi ha portato alla failure. Mi trovo in uno stato erroneo se quello stato mi porta ad una failure. Tale stato potrebbe però essere una combinazione di stati per cui solo mediante tale combinazione si avrebbe una failure. Per essere sicuro di beccare l'errore è necessario avere una sequenza di eventi di sincronizzazione. Devo forzare lo scheduler in modo tale che ogni caso di test mi deve far avanzare gli oggetti in un determinato modo, in modo da testare diverse combinazioni. Usando multi-thread in java si potrebbero avere questi problemi.

CAPITOLO 16

Ispezione

16.1 Introduzione: Ispezione

Tecnica manuale che di solito precede il testing, è applicabile non solo al codice ma anche ai documenti, ed è effettuata sulla base di una checklist. Tale tecnica però è nata sul codice e va a rilevare la presenza di anomalie. Essa è una tecnica completamente manuale e serve per trovare e correggere errori. Non è molto tecnologica anche se si possono usare tecniche di analisi statica in compilazione per rilevare difetti ed è estendibile al progetto, ai requisiti seguendo lo stesso processo. Tecniche di ispezione di codice possono rilevare ed eliminare anomalie fastidiose e rendere più precisi i risultati. L'ispezione avviene mediante:

- **moderatore:** organizza il processo, colui che presiede le sedute, sceglie i partecipanti, controlla il processo
- **lettori, addetti al test:** leggono il codice al gruppo, cercando difetti
- **autore:** colui che ha scritto il codice o il documento. Lui è un partecipante passivo in quanto risponde alle domande quando richiesto ma non fa altro

Se c'è una lettura in pubblico e c'è qualcuno che risponde alle domande, allora c'è un meeting di ispezione.

16.2 Fasi di ispezione

Le fasi sono:

- **pianificazione:** si scelgono partecipanti e checklist e si effettua la pianificazione di meeting. Le checklist sono liste di punti che indicheranno cosa deve essere controllato
- **fasi preliminari:** si forniscono le informazioni necessarie e si assegnano i ruoli.
- **preparazione:** lettura del documento ed individuazione dei difetti
- **ispezione:** ogni ispettore ha individuato i problemi. Nella fase di meeting in cui il codice viene letto a tutti e si vanno ad evidenziare tutti i difetti trovati dai vari ispettori. Non tutti potrebbero essere difetti, mentre altri potrebbero essere stati non trovati in fase preliminare. Alla fine del meeting si fornirà un documento all'utente indicando cosa va cambiato. I difetti vengono evidenziati ma non viene suggerita la soluzione. Come correggere è a competenza dell'autore. Oltre ad avere una checklist l'ispettore ha anche un form in cui si vanno a documentare difetti ed anomalie. La fase di ispezione viene eseguita prima del testing per ridurre il numero di casi di test
- **lavoro a valle:** l'autore modifica il documento per rimuovere i difetti. Si riparte poi con il processo di pianificazione dell'ispezione dei nuovi documenti. La checklist utilizzata in tal caso sarà la stessa

La checklist è un qualcosa che viene data per quel tipo di documento, per cui è sempre la stessa nelle varie ispezioni su quel documento. Nelle riunioni l'obiettivo è trovare il maggior numero possibile di difetti. Si possono fare non più di due riunioni al giorno per non più di due ore, per cui il codice non può essere troppo lungo, in quanto deve essere ispezionato attentamente. Alcuni approcci sono quelli di parafrasare il codice linea per linea cercando di ricostruire l'obiettivo dal codice sorgente. È necessario seguire le checklist, trovare errori ma non correggerli, mentre il moderatore deve evitare che si sforino i tempi prestabiliti.

16.3 Incentivi

Un autore che partecipa al meeting è come se fosse sotto esame, per cui cerca di nascondere i difetti e non evidenziarli, per cui gli errori trovati durante l'ispezione non devono essere usati nella valutazione personale. Errori trovati durante il testing però vengono usati nella valutazione personale.

16.4 Varianti

Alcune varianti sono:

- **revisione attiva di progetto:** un revisore impreparato può sedere in silenzio ed essere inutile. è quindi necessario scegliere revisori con esperienze specifiche. Revisori diversi devono cercare difetti differenti
- **ispezione a fasi:** una serie di fasi più piccole che mettono a fuoco problemi specifici in un ordine determinato. Si avranno ispettori singoli per controlli semplici e più ispettori per controlli complessi

L'ispezione funziona poichè è un processo formale, dettagliato e consente di tracciare il risultato. Le checklist sono parte della progettazione processo di ispezione per cui si può migliorare il processo migliorando le checklist, non essendo statiche. Ci sono inoltre aspetti sociali del processo, ovvero interazione tra ispettore ed autore e si può applicare questa tecnica all'intero progetto, pur non essendo scalabile. Esso si applica anche a programmi incompleti in quanto si applica prima del testing.

16.5 Esempio: Checklist in Java ed in C

Ci sono checklist per:

- **commenti:** vanno a controllare i commenti e la loro correttezza sintattica e ortografica, lo stile ed il contenuto. Per essi viene fatto un controllo sintattico e semantico, se ci sono identificativi, il loro stile come l'uso indiscriminato di abbreviazioni, l'assenza di troppe ripetizioni, il linguaggio comprensibile e frasi leggibili, commento presente in ogni classe e ben visibile, non dispersivi e non densi nel codice il che comprometterebbe la leggibilità. Essi dovrebbero inoltre esprimere i metodi principali della classe, deve dare indicazioni chiare, completo in ogni sua parte e contestuale. Non dovrebbero essere inseriti commenti banali.
- **codice:** controllo sullo stile, correttezza semantica, ridondanza e qualità. Gli identificatori dovrebbero espressi rispettando la convenzione, come per i nomi delle classi. I nomi dovrebbero essere significativi ed il codice non dovrebbe essere né diluito né denso, ma ben indentato. Si fa ispezione semantica e si controlla l'assenza di ridondanze. Si va inoltre a controllare la qualità del codice, ovvero presenza di costruttori,

non si usano metodi con side-effects non necessari, i metodi sono equilibrati e hanno un buon livello di coesione, si fa uso di eccezione e di metodi non deprecati

Checklist simili sono presenti anche per C:

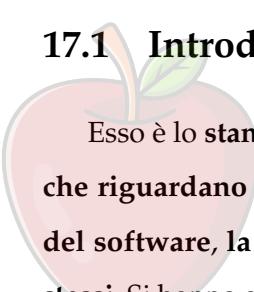
- **rispetto standard di codifica (stile):** Bisogna usare la naming convention, avere nomi significativi. Non è opportuno avere side effects all'interno delle istruzioni. Bisogna controllare ci sono degli spazi con "<, >, =", i files con INCLUDE rispettano lo standard del progetto o se sono gli innested INCLUDE files evitati, uso chiaro di if, else if, switch... usare while rispetto a do-while dove possibile
- **ricerca di potenziali difetti (contenuto):** Conviene avere una funzione con un solo return, altrimenti il codice non è ottimale. Ogni funzione dovrebbe quindi avere un unico punto di ingresso ed un unico punto di uscita. Bisogna implementare l'algoritmo nel modo meno dipendente possibile dal linguaggio.



CAPITOLO 17

Standard IOS/IEC 12207

17.1 Introduzione: ISO/IEC 12207



Esso è lo **standard sui processi di ciclo di vita del software**. Viene utilizzato per processi che riguardano l'acquisizione, la fornitura, lo sviluppo, l'esecuzione, la manutenzione del software, la gestione, il controllo, il miglioramento ed il monitoraggio dei processi stessi. Si hanno alcune differenze con l'ISO/IEC 12207 del 2008. Certificando un'azienda non si sta certificando un singolo progetto, ma l'azienda in generale. Nel manuale di qualità sono definiti tutti gli standard che l'organizzazione utilizza. Quando pianifico la qualità vado a decidere quali processi specifici che utilizzo all'interno di un processo. Gli **standard** quindi per un **progetto** sono definiti per l'intera organizzazione ed eventualmente adattati al singolo progetto. Prima di progettare il software, dobbiamo progettare il software compreso in un sistema. Simulato nell'ambiente simulato, dobbiamo simularlo nell'ambiente reale, poichè è nell'ambiente reale che dovrà essere manutenuto. Importante è la **fase di process establishment** in cui si va definire il **modello di processo da utilizzare**. Si farà poi **process assessment** e **process improvement**. La fase di **process assessment** consiste in un'attività di valutazione e valutazione dei processi all'interno di un'organizzazione al fine di misurarne l'**efficacia, l'efficienza e la conformità agli standard o alle best practice**. La fase di **process improvement** consiste invece in un concetto e un'attività che **mira a rendere i processi organizzativi più efficienti, efficaci e allineati agli obiettivi aziendali**.

17.2 Principi di base

Lo standard ISO12207 ha una struttura che si basa sui seguenti **principi**:

- **ingegneristici**
- **indipendenza dal modello di un ciclo di vita del software**
- Ha a che fare con **total quality management**, in quanto per **ogni processo viene definito un ciclo "plan-do-check-out"** (PDCA)
- **tailoring**; adattabilità a contesti e organizzazioni diverse
- **criteri di collaborazione cliente-fornitore**
- **gestione della documentazione**

17.3 Architettura e processi

L'architettura del ciclo di vita è basato su un insieme di processi e interrelazioni tra di essi. Si fonda su principi di coesione e responsabilità. Ogni processo è descritto in termini di: **titolo, scopo, output, task ed attività**. Un'attività è un insieme di task coesi che prendono un input e producono un output.

Per ogni processo, oltre alla lista delle attività e dei task, lo standard riporta gli obiettivi e le responsabilità. Lo standard non definisce un ordine di esecuzione di processi, attività e task. I processi hanno delle attività ma il modo di condurre le attività dipende dal modello di processo che non è definito dallo standard ma dall'**organizzazione che adotta lo standard**.

17.4 Classi di processi

Esistono vari tipi di processi:

- **processi primari**: hanno a che fare con i **processi che direttamente servono all'acquisizione, fornitura sviluppo manutenzione ed esercizio**
- **processi di supporto**: sono di supporto ai processi primari
- **processi di organizzazione**: sono utilizzati per stabilire, controllare e migliorare un **processo del ciclo di vita**

Alle 3 categorie è possibile aggiungere un **processo di tailoring** per adattare lo standard alla particolare organizzazione.

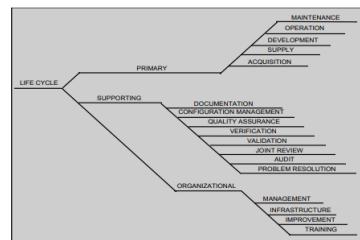


Figura 17.1: Classi di processi

17.5 Processo di agreement

Definiscono le attività necessarie per **stabilire un accordo tra due organizzazioni**:

- **acquisition process:** Lo scopo è quello di **ottenere un prodotto e/o un servizio che soddisfa le necessità dell'acquirente**
- **supply process:** Lo scopo è quello di **fornire un prodotto o un servizio all'acquirente che soddisfa i requisiti concordati**

Acquisizione e fornitura sono due processi di agreement tra l'utilizzatore e coloro che sviluppano il software.

17.5.1 Acquisition process

Le attività sono:

- **acquisition preparation:** il cliente deve preparare la gara d'appalto per acquisire software o in caso capire cosa dovrebbe fare il software
- **acquisition advertisement:** facendo una gara, essa deve essere pubblicizzata
- **supplier selection:** si sceglie il fornitore
- **contract agreement:** si effettua il contratto
- **agreement monitoring:** si monitora il contratto
- **acquirer acceptance:** accettazione della nuova parte del sistema rilasciata
- **closure:** chiusura del contratto

Le operazioni non vengono fatte in modo lineare, ma alcune fasi potranno essere iterative

17.6 Supply process

Le attività sono:

- **Opportunity Identification:** c'è il **commerciale che sta in ascolto o osserva possibili gare** a cui partecipare
- **Supplier Tendering:** si fa un'offerta, **si partecipa alla gara**
- **Contract Agreement:** **vinco la gara per cui faccio un contratto**
- **Contract Execution:** chi esegue il **contratto monitora che tutto vada bene**. Colui che **monitora è il commerciale**. Bisogna stare attenti a rispettare le scadenze definite nel contratto
- **Product/Service Delivery and Support:** se accetto, **devo rilasciare e supportare il rilascio del sistema**
- **Closure:** chiusura del contratto

Ciò potrebbe essere anche un **processo di manutenzione**. Ciò che **cambia è come avviene il servizio e come è rilasciato il sistema**.

17.7 Organizational Project-Enabling Processes

Riguardano la **capacità di un'organizzazione di acquisire o fornire prodotti o servizi attraverso l'iniziazione, il supporto e il controllo di progetti**. Essa è composta da processi come:

- **Life Cycle Model Management Process:** Ha l'obiettivo di definire, mantenere e assicurare la **disponibilità di policy, life cycle processes, life cycle models e procedure** consistenti con gli obiettivi aziendali e capaci di supportare le esigenze dei singoli progetti. Le attività sono: **Process establishment, Process assessment e Process Improvement**
- **Infrastructure Management Process:** Ha l'obiettivo di **fornire le infrastrutture e i servizi abilitanti** (facility, tool e ICT asset) **per i progetti** dell'organizzazione. Le attività sono: **Process implementation, Establishment of the infrastructure e Maintenance of the infrastructure**
- **project portfolio management process:** Il processo viene iniziato nella fase di "project portfolio management", per cui viene iniziato in questa fase, **viene valutato il portfolio**

e la chiusura del progetto. Nella valutazione del portfolio si va a valutare la qualità del portfolio dei progetti. Si ha un buon portfolio quando si ha un buon bilanciamento di progetti in varie fasi.

- **human resource management:** Ha l'obiettivo di fornire all'organizzazione le risorse umane necessarie e mantenere le loro competenze consistenti con le necessità di business. Per fare ciò ho bisogno di persone diverse per cui le attività sono di skill identification, skill development, Skill acquisition and provision. Per fare ciò è necessario il knowledge management, ovvero bisogna gestire la conoscenza dell'azienda a livello di asset o risorse umane...
- **Quality Management Process:** Ha l'obiettivo di assicurare che prodotti, servizi e implementazioni dei processi del ciclo di vita siano aderenti agli obiettivi di qualità aziendali e soddisfino il cliente. Esistono inoltre dei progetti per la gestione ed il supporto ai processi

17.8 Altri tipi processi

Altri tipi di processi sono:

- **Technical Processes:** usati per definire i requisiti del sistema per trasformarli in un prodotto, per permettere la riproduzione consistente di un prodotto, per usare il prodotto, per fornire i servizi richiesti, per consentire il ritiro del prodotto. Essi partono dalla fase di requirement elicitation, essendo però a livello di sistema
- **Software Disposal Process:** Ha l'obiettivo di terminare l'esistenza di un'entità software di un sistema. Il processo termina il supporto attivo di esercizio e manutenzione o disattiva, **disassembla e rimuove i prodotti software coinvolti**, lasciando l'ambiente target in una condizione accettabile
- **Software Documentation Management Process:** Una specializzazione dell'Information Management Process con l'obiettivo di sviluppare e mantenere le informazioni software prodotte
- **Software Review Process:** Ha l'obiettivo di mantenere una comprensione comune con gli stackholder dei progressi rispetto agli obiettivi dell'accordo e di cosa dovrebbe essere fatto per assicurare lo sviluppo di un prodotto che soddisfi gli stackholder

- **Software Audit Process:** Ha l'obiettivo di consentire ad una parte indipendente di determinare l'aderenza di prodotti e processi selezionati rispetto a requisiti, piani, o accordi

17.9 Software Reuse Processes

Supportano l'abilità di un **organizzazione** a riusare **software items** oltre i confini di un **progetto**. Esso si divide in:

- **Domain Engineering Process:** ha l'obiettivo di **sviluppare e mantenere modelli, architetture e asset di dominio**. Ha il compito di **capire il dominio, per costruire componenti software riusabili** in vari domini
- **Reuse Asset Management Process:** ha l'obiettivo di **gestire il ciclo di vita degli asset riusabili dalla loro concezione al loro ritiro**. Il **riuso è legato al particolare dominio in cui mi trovo**. Per poter riusare devo però mettere in piedi un sistema di archiviazione di tali componenti
- **Reuse Program Management Process:** ha l'obiettivo di **pianificare, stabilire, gestire, controllare e monitorare un programma di riuso di un'organizzazione e di sfruttare sistematicamente opportunità di riuso**.

CAPITOLO 18

Project management

18.1 Introduzione: Project management

Anche il project management si serve di processi che fanno parte dei processi del ciclo di vita del software. Il **project management** è importante poichè è necessario qualcuno che **amministri in modo efficiente**. Il **management** è definito in termini di funzioni: **pianificazione delle attività, organizzazione di risorse per svolgere le attività, direzione, controllo e comunicazione**. La **pianificazione** è un'attività continua che viene fatta periodicamente. Un manager ottiene fondamentalmente dei **risultati** facendo lavorare gli altri, **organizzando e controllando il lavoro degli altri**, riuscendo quindi a fare cose attraverso le persone. Il **project management** invece è definito come il **management nel contesto di un progetto**. Il progetto ha delle **caratteristiche peculiari** in termini di costi, tempi e vincoli esterni, per cui si cerca di **compiere uno specifico task in un time frame specifico e con risorse definite**. Una cattiva organizzazione può determinare l'intero fallimento di un progetto. Si fa quindi una distinzione tra **management** e **project management**:

- **management:** nel management si hanno degli obiettivi.
- **project management:** contesto in cui ci sono dei vincoli specifici.

Esempio: avendo degli obiettivi di budget, definendo che una determinata organizzazione deve fatturare tot soldi, è necessario che tale obiettivo venga raggiunto. L'organizzazione alle spalle servirà a far rispettare i tempi di consegna... L'obiettivo però non è rispetto un

qualcosa da realizzare come progetto, ma l'obiettivo è quello di realizzare un prodotto o un servizio entro certi tempi ed entro certi costi. L'obiettivo è riuscire a produrre il prodotto o servizio entro certi vincoli, mentre nel process management l'obiettivo è di far guadagnare un tot di soldi.

18.2 Software project management

Esso riguarda **attività che assicurano che il software sia rilasciato in tempo rispetto lo schedule ed in accordo con i requisiti dell'organizzazione** che produce e sviluppa il software. Il project management è necessario quindi poichè **si hanno sempre dei vincoli di budget e di schedule da rispettare**. Se si vuole **realizzare un prodotto che non sia software** **ho caratteristiche di management differenti**. Le tecniche in realtà saranno più o meno le stesse ma **il tipo di prodotto andrà a condizionare il modo in cui esse possono essere applicate**:

- **il prodotto è intangibile**, a differenza del software
- **il prodotto è flessibile**, a differenza del software
- **la software engineering non è riconosciuta come una disciplina ingegneristica**
- **il processo di sviluppo del software non è standardizzato**
 - molti **progetti software hanno aspetti peculiari**. Nel caso della realizzazione di prodotti, il processo che vado ad applicare, essendo standardizzato, è più o meno lo stesso in quanto il prodotto non cambia molto, mentre con il software è l'opposto. **Spesso si creeranno software unici e quindi differenti dai precedenti creati**

18.3 Management activities

Ci sono diverse attività di management che un manager si trova ad affrontare:

- **proposal writing**: immaginando di voler partecipare ad una gara d'appalto, **nella risposta bisogna scrivere certe cose e come esse vengono fatte**. La parte tecnica della risposta deve essere fatta dal commerciale in accordo con un project manager, che va a gestire l'aspetto tecnico del progetto che si vuole realizzare. **Ciò accade anche se si ha un'idea di prodotto da proporre a qualcuno con cariche maggiori delle mie**. Fondamentale per chi si occupa di project management

- **project planning and scheduling:** una volta che la proposta viene approvata bisogna iniziare a pianificare il progetto, andando a schedulare le attività
- **project costing:** assegnare i costi alle attività, stimare i costi delle varie attività
- **Project monitoring and reviews:** controllare le attività. In essa è contenuta anche la project execution in cui il project manager assegna le attività a diversi team. All'interno del team si potrebbe scegliere in modo autonomo come assegnare i task. In casi in cui il progetto sia grande, non sarà il project manager a gestire tutti i dettagli, ma si interfacerà con il team di management che a sua volta gestisce altre persone. I task verranno assegnati dall'alto verso il basso ed i report passeranno dal basso verso l'alto
- **Personnel selection and evaluation:** immaginando di dover assumere delle persone, è necessaria una struttura del personale. Tale personale dovrà essere valutato sulla base di capacità, skill di interazione... ma non potranno valutare le capacità tecniche, compito che dovrà essere svolto dai project manager. Si andranno inoltre a valutare le persone
- **Report writing and presentations:** il project manager riporta i risultati al top manager o ai stackholder del progetto. Sarà quindi necessario fare rapporto a loro.

Tali attività non sono peculiari del software, ma sono attività applicabili a una realizzazione di qualsiasi progetto. Ciò che cambia è il tipo di prodotto

18.4 Progetto software

Insieme di tutte le attività tecniche e manageriali richieste per sviluppare e rilasciare i deliverables al cliente. Il progetto ha una specifica durata, consumare risorse e produce un work product. Le attività di management che servono a completare un certo software sono task, attività e funzioni. Un'attività è decomponibile in sotto-attività fino ad arrivare ai task, ovvero le parti non decomponibili della gerarchia. Le funzioni invece sono attività di progetto che dura per tutto il progetto. Project management, quality management e configuration management sono ad esempio funzioni di progetto. Queste funzioni possono essere composte da attività ma comunque sono attività che durano per l'intero progetto

18.5 Software Project Management Plan (SPMP)

Documento di controllo per un progetto. Specifica tutti gli approcci **tecnicici e manageriali** che vengono utilizzati per sviluppare il software. Di solito tale documento va **insieme al documento di analisi dei requisiti**. Fondamentalmente, quando si sviluppa un progetto, all'inizio non posso pianificare nel dettaglio tutte le attività, poichè **non sono ancora chiari quali siano i requisiti, quali siano i casi d'uso, quale sia il sottosistema da implementare...** per cui si pianifica uno **schedule delle attività non nel dettaglio**. Successivamente si **definiscono i requisiti, casi d'uso avendo un'idea di quali saranno quindi i sottosistemi**. Ciò comporta un'**organizzazione in team** che può variare sulla base dei requisiti, per cui non si può pianificare in dettaglio finchè **non ho almeno i requisiti ed i casi d'uso**. I requisiti definiscono in un aspetto fondamentale del sistema: **lo scope**.

SPMP può essere parte di un **project agreement**, in quanto si potrebbe commissionare a qualcuno solo l'**analisi dei requisiti**, volendo sapere se posso realizzare un certo progetto. Data l'**analisi dei requisiti** è necessario **definire in quanto tempo e con quali costi tale progetto possa essere realizzato**. Si può poi successivamente **arrivare ad un accordo per lo sviluppo di tale progetto sulla base dell'SPMP fornito**.

18.6 Componenti di un progetto

Un progetto è composto da 4 componenti: **work product, schedule, task e partecipanti**. È possibile vedere tale struttura anche in modo più complesso. In tal caso si parla di attività suddivise in task, dell'unità organizzativa che può essere formata da partecipanti o staff... Tale organizzazione presenta meglio un'**organizzazione gerarchica delle risorse**.

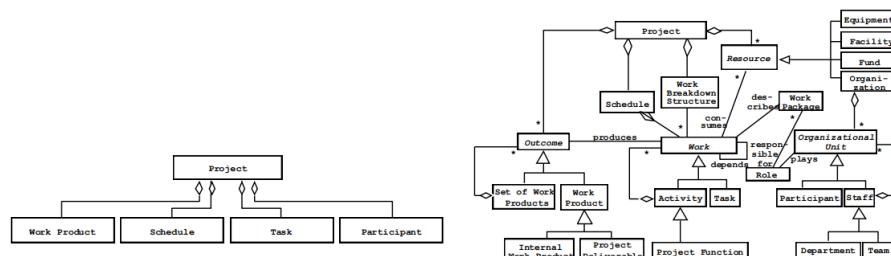


Figura 18.1: Componenti di un progetto

18.7 Stati di un progetto

I vari stati sono:

- **conception:** si ha la **formulazione dell'idea**, andando ad eseguire **benefit analysis e feasibility study**
- **project definition:** si definisce il **problem statement**, si definisce l'**architettura del software** e si definisce un **piano**
- **start:** si parte con il **setup dell'infrastruttura, skill identification, formazione del team e project kickoff**
- **steady state:** si sviluppa il sistema, si effettuano controlli, si effettuano **risk management**. Potrebbero **emergere nuove necessità o tecnologie** per cui si potrebbe fare **replanning**, tornando alla fase di conception. Avendo **realizzato quanto definito nello scope**, posso passare alla fase di **termination**
- **termination:** si effettua la **client acceptance, il delivery e post mortem**, ovvero si analizza il progetto per fare **lessons learned**, vedere cosa è andato bene e cosa è andato male, basandoci sui dati. A volte **non lo si fa su singoli progetti ma su più progetti per capire complessivamente come stia andando l'azienda**, per capire se ci sia qualcosa da migliorare a livello aziendale, se si hanno problemi a livello di risorse, tecnologie...

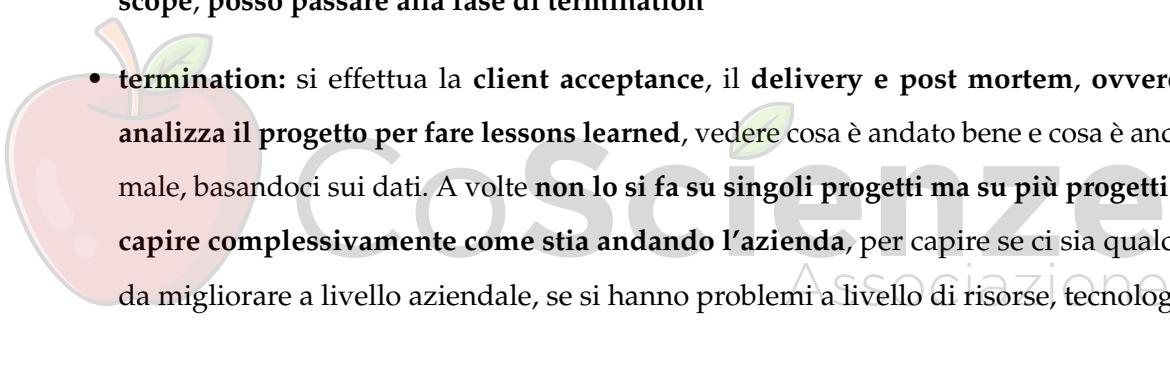


Figura 18.2: Stati di un progetto

18.8 Aspetti di un progetto

: Ci sono 3 aspetti di un progetto: **scope**, ovvero quello che devo realizzare a livello di requisiti, **tempi** e **costi**. La **pianificazione delle attività** dipende dallo **scope** e lo **scope** a sua volta è legato ai requisiti. Cambiando i requisiti cambia lo **scope** e cambiando lo **scope** cambia la **pianificazione temporale** delle attività. I costi non cambiano molto in quanto bisogna comunque rientrare nel **budget predefinito**. La **tempistica**, quindi data di rilascio, e

costi sono definiti. Cambiando lo scope devo comunque adattare la mia organizzazione in modo tale da riuscire a produrre quanto definito nello scope ma rispettando tempi e costi. Non è detto che ciò sia fattibile. I costi complessivi cambiano in base a come sono definite le attività. Mettendo più persone a lavorare su un'attività riduco i tempi ma aumento i costi. L'idea è di trovare il miglior schedule per il progetto in modo da rispettare tempi e costi, realizzando ciò che è definito nello scope. Ogni modifica allo scope provoca una modifica al piano.

18.9 Project agreement

Il project agreement può essere un contratto, business plan o un project charter. Quest'ultimo è un documento che definisce cosa debba essere realizzato. Documento scritto per il cliente che definisce: scope, durata, costi e deliverables del progetto. Esso contiene i requisiti che vanno a definire lo scope e la pianificazione del progetto. Definisce inoltre gli item esatti, le quantità, le date dei deliverables e la delivery location. I clienti sono individui singoli o organizzazioni che specificano i requisiti e accettano il piano di progetto. I deliverables sono i work product rilasciati al cliente possono essere: documenti, dimostrazioni di funzioni, dimostrazioni di requisti non funzionali o dimostrazioni di sottosistemi.

18.10 IEEE 1058

Lo standard per l'SPMP è lo standard IEEE 1058. Quello che fa è:

- specificare il formato ed il contenuto di project management plan
- fornisce lo standard di astrazione per il project manager o per l'intera organizzazione per costruire un insieme di pratiche e procedure per sviluppare un SPMP.

Esso non definisce il modo in cui i processi vengono realizzati, ma ci dice quali sono le procedure e com'è l'organizzazione dell'SPMP. Non specifica inoltre come le procedure e tecniche devono essere utilizzate e non provvede esempi

18.11 Project agreement, problem statement e SPMP

Il problem statement è definito dal client o dallo sponsor. Esso viene fornito al project manager e al team che realizza l'SPMP, definisce tempi e costi del progetto. Tali informazioni tornano indietro sotto forma di project agreement che viene poi accettato dal cliente.

Se non ci fosse una **stima di tempi, costi e attività** che devono essere realizzate, quindi un **SPMP preliminare**, non posso fornire un **project agreement al cliente**.

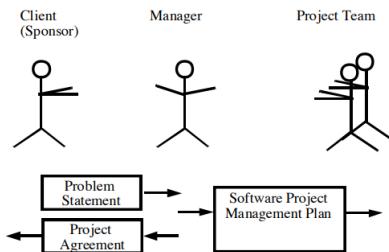


Figura 18.3: Project agreement, problem statement e SPMP

18.12 Organizzazione di un SPMP

Esso è organizzato in:

- **Front Matter:** formato da **titolo**, l'**update history** in cui vengono riportati i cambiamenti, lo **scope e purposes** dello schema, **table of contest**, **figure e tavelle**
- **Introduction:** project overview, i **deliverables** con relative scadenze, **piani per l'evoluzione** dell'SPMP anche per evoluzioni non anticipate. Nel caso si verifichi un'**evoluzione non anticipata**, ovvero modifiche per cui non so cosa dovrà esserci, **dovrò definire come gestire le cose**. Si avrà inoltre un **materiale di riferimento** e la **definizione di acronimi**
- **Project Organization:** in questa fase **non si definisce il modello di processo** dello **sviluppo software**, ma il **modello di processo** afferente alla **fase di management**. Si definisce la **struttura organizzativa**, l'**interfaccia organizzativa** che esprime le **relazioni con altre entità** come fornitori.... e il **responsabile del progetto** ovvero le principali funzioni ed attività, quale ruolo si deve occupare di ciò. è inoltre presente la **matrice di progetto funzioni/attività**
- **Managerial Process:** Essa contiene **priorità ed obbiettivi del managment**. In questa fase vengono **dettagliati i processi manageriali**. Assunzioni, **dipendenze e vincoli**, la parte di **risk managamenent** (identificazione, meccanismi per tracciare i rischi e piani di contingenza). Ci sono meccanismi di **monitoring e controll e lo staffing plan** (quali persone, quante persone mi servono e come le acquisisco).

- **Technical Process:** si fa riferimento a **metodi, tool e tecniche**, potendo fare riferimento ai **piani per la documentazione, piani per assicurare la quality assurance**
- **Work Elements, Schedule, Budget:** Essa contiene la **WBS**, ovvero la **work breakdown structure**. Essa è una **decomposizione gerarchica di un progetto in attività e tasks**. È poi necessaria definire la sequenza temporale, andando a definire le **dipendenze tra tasks**. Devo quindi definire il **tempo impiegato per tali attività** ma anche l'ordine in cui esse si susseguono. Sarà inoltre necessario gestire le persone in quanto è probabile che un'attività sia svolta più velocemente se eseguita da più persone. Sarà quindi necessario definire il **budget**, il **tipo di persone**, i **requisiti della mantenance e del training**, il budget ed il costo stimato per ogni task. I costi ed i tempi sono legati.
- **Optional Inclusions**

Nel caso del **software si parla di diseconomia di stato**, ovvero **aumentando la dimensione del prodotto, il costo per realizzare il prodotto aumenta**. L'economia di scala dice invece che più produco più risparmio. Per produrre quindi **progetti di dimensione maggiore** si cerca **ci mantenere i tempi per cui si aumenta il costo**. Una volta che ho il tempo per le attività **potrò capire quanto tempo ci metto a realizzare il progetto** e se riesco a mantenere i vincoli. Per questo è un'attività iterativa, ovvero perchè bisogna **rispettare vincoli e fare qualcosa in caso non riesca a rispettarli**.

18.13 Risk management

La **risk management** riguarda l'**individuazione di rischi**, capire quale sia la **probabilità con cui un rischio può verificarsi** e quale sia il suo **impatto**. Deve inoltre produrre dei **piani o per evitare il rischio e ridurre la probabilità che esso si verifichi, o ridurre l'impatto di tale rischio**. Il **rischio** è quindi la **probabilità che qualcosa vada storto**. Ci sono **rischi positivi** in quanto il progetto potrebbe darmi un qualcosa di inaspettato. In tal caso si cercherà di far avverare tale rischio e di amplificarne l'effetto. I **rischi possono essere legati al progetto, influenzando schedule o risorse, o potrebbero avere impatto sulle risorse, influenzandone la qualità o potrebbero impattare il business**. I **rischi provengono dal prodotto**.

18.13.1 Fattori di rischio nel software project

Alcuni rischi possibili sono:

- **rischi contrattuali:** cosa succede se un cliente fallisce? Per gestire tale rischio si potrebbe optare per uno sviluppo incrementale, per cui rilascio poco alla volta e vengo pagato poco alla volta
- **rischi legati alla size del progetto:** il progetto potrebbe essere più grande di quanto stimato
- **complessità del progetto:** la complessità potrebbe essere maggiore di quanto stimato
- **personale:** dipende sia da come assumo sia dal fatto che qualcuno potrebbe lasciare il progetto. Con la project stuffing potrebbero esserci dei problemi in quanto il budget non mi consente di prendere persone con una certa esperienza, oppure non ci sia disponibilità di persone con una conoscenza adeguata o si potrebbero avere dei vincoli di assunzione posti dall'azienda.
- **client acceptance:** il cliente potrebbe non accettare quanto proposto

18.13.2 Processo di risk management

Il processo di risk management è suddiviso in 4 fasi:

- **risk identification:** La prima cosa da fare è individuare la lista dei possibili rischi. Si individua quindi il rischio, la sorgente del rischio ed i suoi effetti. Alcuni rischi potrebbero essere legati alla:
 - **tecnologia:** il software non riesce a reggere il passo delle transazioni
 - **people:** impossibile effettuare il recruiting dello staff con determinate skills
 - **organizzazione:** l'organizzazione è ristrutturata in modo tale da avere diversi responsabili per il progetto. Ciò potrebbe essere dovuto al cambio del responsabile del progetto
 - **tools:** CASE tools sono inefficienti e non possono essere integrati
 - **requirements:** cambio dei requisiti che richiedono major rework, o l'utente non capisce l'impatto della modifica di requisiti
 - **stime:** stime sbagliate nella grandezza del software e nel tempo necessario per lo sviluppo
- **risk analysis:** per ognuno dei rischi la probabilità che tale rischio possa accadere e l'effetto che esso può avere. Tale fase viene fatta per capire probabilità ed impatto

di ogni rischio. Si fa una **scala ordinata che andrà a rappresentare la probabilità che un rischio si verifichi**. Lo stesso viene fatto per l'**effetto del rischio**, andando da non interessante a catastrofico. Per capire se ci sono problemi legati ad esempio alla **tecnologia**, si potrebbe vedere se ci sono **molte lamentele sulla funzionalità del software**. Il project manager deve gestire i conflitti all'interno del progetto.

- **risk planning:** Fatta l'analisi e prioritizzati i rischi è necessario predisporre i piani per gestire i rischi. Avendo un rischio con **probabilità bassa ma con effetto catastrofico**, devo decidere il piano di contingenza vado a mettere in pratica. Esso potrebbe essere sia un **piano di avoidance che di minimization**. Poichè la probabilità è bassa però non si va a spendere molto in pratica per minimizzare ma piuttosto **si cerca di diminuire la probabilità che tale rischio si verifichi**. Il **piano di contingenza è un piano che viene stipulato a posteriori per operare sul rischio**. Come possibili soluzioni a **personale che se ne va**, si potrebbe utilizzare l'**overlap**, che ha però dei costi
- **risk monitoring:** si vanno a **monitorare i rischi attraverso il progetto**. Mettendo in pratica le azioni vado a verificare che tali azioni abbiano avuto degli effetti sul sistema. I rischi da monitorare sono gli stessi della fase di pianificazione. Di seguito sono riportatigli indicatori potenziali per tali problemi:
 - **technology:** ci sono vari problemi afferenti al **software**
 - **people:** il **morale è basso** o ci sono dei problemi tra gli sviluppatori
 - **organizzazione:** presenza di gossip e mancanza di un senior manager
 - **tools:** i **team members sono riluttanti nell'uso di tool**
 - **requisiti:** ci sono **molte richieste di requirements change request**
 - **stima:** faliure nel definire con chiarezza i difetti o failure nel raggiungere schedule che erano stati accordati

Il ciclo è continuo in quanto dal monitoraggio posso andare a fare assessment dei rischi. Il piano deve inoltre essere attuato. Se voglio evitare che un qualcosa si verifichi, devo mettere in piedi azioni per evitare che essi si verifichino. Si va poi a vedere se tale pianificazione ha funzionato mediante il monitoring.

Esempio: supponendo che una persona importante lasci il progetto, quello che devo fare è sviluppare un piano. Mettere in atto piani per la gestione dei rischi ha dei costi. I piani e le azioni sono sempre preventive. I piani che metto in atto successivamente al problema, sono

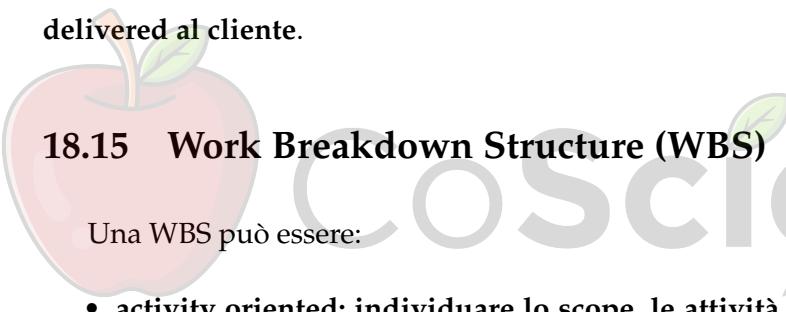
definiti piani di contingenza. non posso evitare che cambi il top manager dell'organizzazione, per cui quello che si può fare è ridurre l'impatto



Figura 18.4: Risk management steps

18.14 Work package e work product

Un **work package** è una specifica per il lavoro che deve essere compiuto in un'attività **un task**. Il **work product** è l'item risultato da una funzione, attività o task. Il **project baseline** è **work product** che è stato formalmente rivisto e approvato e non può essere modificato se non attraverso una richiesta formale di modifica. Un **project deliverable** è la versione **delivered al cliente**.



18.15 Work Breakdown Structure (WBS)

Una WBS può essere:

- **activity oriented:** individuare lo **scope**, le attività, gli elementi che devono essere prodotti, è simile alla fase di **requirement elicitation**, dovendo andare ad **organizzare** tali attività all'interno della WBS.
- **product oriented:** individuare il **prodotto** che deve essere eseguito, suddiviso a sua volta in attività. Permette di definire il modo in cui organizzare il prodotto. Si definisce un'attività e si ripete quella

Di solito **conviene utilizzare entrambi gli approcci**, per capire sia quali sono le attività da svolgere che il modo di organizzare il prodotto in modo da incastrare attività e prodotto. Il **costo per pianificare una WBS** dipende dal progetto. Maggiore sarà il numero di mesi/uomo maggiore sarà il tempo per la pianificazione.

18.15.1 Da WBS a Dependency graph

La **WBS** non mostra alcuna dipendenza temporale tra le attività e tasks. è necessario definire attività che devono essere svolte prima di altre o se ci sono attività che possono

essere svolte in parallelo. Quello che si fa è quindi minimizzare le dipendenze tra task in modo da evitare delay causati da un task che aspetta un altro o si organizzano i task in modo da fare un uso ottimale della forza lavoro. Potrebbe essere necessario identificare attività critiche, ovvero attività che se rallentate rallentano l'intero processo. Tale ricerca viene fatta mediante critical path analysis. Le dipendenze temporali tra attività possono essere espresse mediante dependency graph, in cui i nodi sono attività e gli archi sono le dipendenze temporali. Potrebbe essere complesso organizzare il lavoro senza dependency graph in caso in cui ci siano molte cose da fare.

18.16 PERT Chart

Ogni attività ha un **tempo di inizio** e una durata stimata. Tutte le attività segnate in rosso hanno slack time pari a 0. Lo slack time indica il tempo in cui posso ritardare un'attività senza ritardare il progetto. Nel caso non venga rispettato lo slack time, il progetto potrebbe essere ritardato. L'obiettivo del PERT è quindi determinare la durata totale del progetto, determinare i critical path e determinare gli slack time.

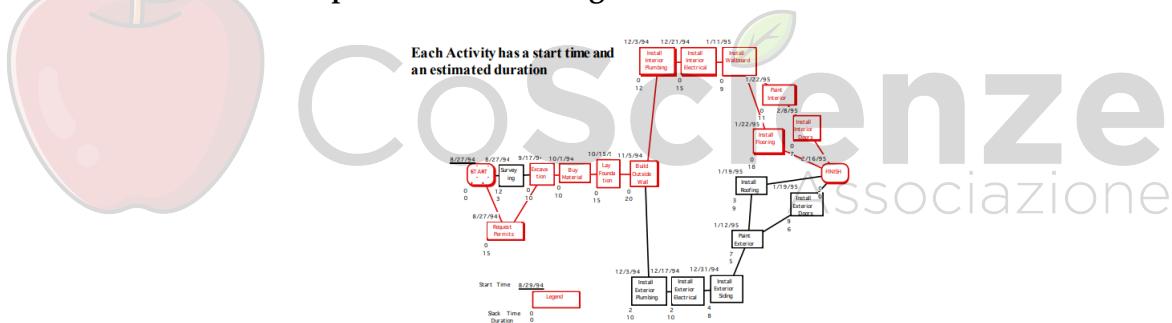


Figura 18.5: PERT Chart

18.17 Dependency diagram

Il dependency diagram contiene:

- **eventi:** le milestones
- **attività:**
- **span time:** durata delle attività che dipendono dalla disponibilità delle persone, dal livello di parallelizzazione, la disponibilità delle risorse

Il dependency diagram può essere disegnato in due modi: activity on the arrow e activity in the node. Nel primo caso i nodi sono le milestones e gli archi sono le attività. Per gli umani è molto più comprensibile l'activity in the node. Dovendo usare però un algoritmo che calcola la durata del progetto basandosi sul tempo delle attività, l'activity on the arrow è un grafo più semplice da utilizzare. Anche PERT usa questo tipo di notazione, usando un diagramma delle attività dove però le attività sono sugli archi.

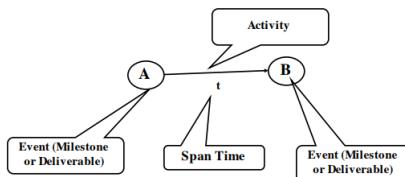


Figura 18.6: Dependency diagram

18.18 Activity diagram

Una volta stimato il tempo di ogni attività ed organizzato in un'activity diagram, per calcolare la durata del progetto possiamo usare un algoritmo basato su due tecniche:

- **forward pass:** Quello che so è quando il progetto parte. L'attività che si sussegue avrà data di inizio pari all'attività di fine dell'attività precedente. Quello che posso fare quindi è calcolare l'early start date e early finish date di ogni attività riuscendo a calcolare l'early start date e early finish date del progetto. Esso determina il critical path. Durante il calcolo del critical path la latest start date sarà pari all'earliest finish date. Nel caso in cui si abbia un'attività che deve aspettare la terminazione di due, l'earliest start date dell'attività sarà pari all'earliest finish date dell'attività che termina per ultima
- **backward pass:** esso determina lo slack time. Per le attività sul percorso critico non si troverà mai differenza tra latest start date e latest finish date. Quelle che non si trovano sul percorso critico avranno invece una earliest finish date minore alla latest finish date

Il critical path è la sequenza di attività che prende il tempo più lungo per essere eseguito. La lunghezza del critical path mi dà la durata minima del progetto. Meno di quel tempo non posso impiegare a meno che il progetto non venga riorganizzato. Esso indica il tempo minimo perché allungando il tempo di quelle attività il progetto durerà di più. Il noncritical

path è la sequenza di attività che possono essere ritardate senza compromettere la durata del progetto. Lo slack time è il tempo massimo in cui le attività possono essere ritardate finendo però comunque in tempo. Gli slack time delle attività non sono indipendenti. Usando lo slack time per un'attività, non posso usare lo slack time anche per l'attività successiva, ma devo ridurre lo slack time dell'attività successiva.

18.19 Action item

Un' action è un task assegnato ad un partecipante. Essa deve rispondere alle domande "what, who, when". Se manca una di esse non è un' action item. Esse dovrebbero comparire nel report e nel meeting agenda.

18.20 Conclusioni: Leggi di progetto

Alcune "leggi" sono:

- i processi evolvono velocemente fino ad arrivare al 90%. Rimarranno incompleti per sempre
- quando le cose stanno andando bene, qualcosa andrà male
- se le cose possono andare male ci andranno
- se le cose sembrano andar bene, qualcosa sta andando male
- Se si permette al contenuto di progetto di cambiare liberamente, il rate di cambiamento eccederà il rate di progresso.
- i team di progetto detestano il progress report poichè manifesta l'assenza di progresso

18.21 Problemi di scheduling

Alcuni problemi potrebbero essere:

- stimare la difficoltà di un problema ed il costo per stimare la soluzione è molto difficile
- la produttività non è proporzionale alle persone che lavorano su un task

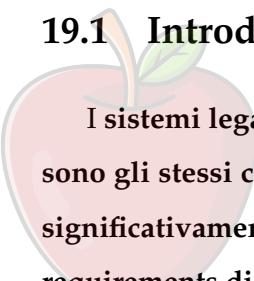
- aggiungere persone tardi ad un progetto potrebbe creare di problemi di comunicazione
- l'impossibile può sempre accadere. I piani di contingenza sono sempre necessari



CAPITOLO 19

Legacy system

19.1 Introduzione: legacy systems



I sistemi legacy sono sistemi ereditati, in quanto coloro che lo hanno sviluppato non sono gli stessi che lo utilizzano. Un sistema legacy è un sistema informatico che resiste significativamente alle modifiche e all'evoluzione che servono ad andare incontro ai requirements di business che cambiano. Esso è quindi un sistema difficile da far evolvere. Essi sono grandi sistemi software difficili da gestire ma che sono vitali per le aziende. Essi sono complessi da gestire in quanto ereditati, ovvero l'azienda che usa il sistema non è attualmente composta dalle stesse persone che hanno sviluppato il sistema stesso, per cui manutenzione e gestione è fatta da esterni a cui le aziende si fidano.

Anche per la manutenzione e gestione, pur essendoci personale fidato all'interno dell'azienda, si preferisce affidarsi ad esterni specializzati in tali operazioni. Spesso un sistema legacy è vecchio di circa 30/40 anni, scritto in un linguaggio obsoleto, ha milioni di linee di codice, si ha poca coesione nei documenti, scarsa gestione dei dati, molto difficile ma non impossibile da espandere e funzionano su vecchi processori.

19.2 LIS e decomponibilità

Molto importante è la decomponibilità dei LIS, avendo user interface, business logic e data management completamente separate. Un sistema decomponibile è quando si ha

solo la user interface separata. Un sistema è non decomponibile quando non si ha alcuna divisione. L'idea è quindi in caso quella di renderlo prima semi-decomponibile e poi wrapparlo. Ciò però non è facile come operazione.

19.3 Soluzione per i Legacy Information Systems (LIS)

Esistono diverse soluzioni per i LIS:

- **freeze:** i sistemi non evolvono più per cui vengono incapsulati, venendo utilizzati così come sono.
- **outsource:** la manutenzione è fatta da un'azienda di terze parti o costruisco o compro un prodotto che possa sostituire il sistema legacy
- **wrapper:** si continua ad utilizzare il sistema ma lo si incapsula, ovvero lo inserisco all'interno di un sistema più ampio che sto sviluppando. Tutta la parte nuova di sviluppo è esterna al sistema wrappando le funzionalità del sistema che continuo ad usare. Wrapping e freeze possono essere usate insieme, usando le funzionalità del vecchio sistema che non si evolve più mediante un wrapper.
- **migrate:** effettuando la migrazione non si esclude il wrapping, in quanto si può migrare wrappando prima le componenti. Incrementalmente, il sistema legacy viene wrappato verso la nuova piattaforma.

19.4 Wrapping

Wrapping significa incapsulare il vecchio sistema legacy con un nuovo software layer in modo da fornire una nuova interfaccia e nascondendo la complessità del LIS. Il layer di incapsulamento può comunicare con il componente LIS mediante socket, procedure remote (RPCs) o mediante interfacce (API). Il sistema wrappato è quindi visto come un server remoto che fornisce servizi ad un client che non ha idea di come i servizi siano implementati. L'interfaccia esterna deve quindi essere compatibile con il client utilizzato. Ciò che c'è dietro invece, potrebbe essere qualunque cosa che permetta al wrapper di comunicare con il LIS. Il concetto di wrapper che usa un componente esterno è molto simile al concetto di adapter, questo poiché si ha un'interfaccia che viene offerta e che deve essere convertita all'oggetto legacy. Magari è necessario combinare più operazioni del sistema legacy per fare ciò che il client vuole. Viceversa, dovendo wrappare a livello di user interface, la cosa si fa

complessa in quanto l'adapter sincronizza solo client e LIS. Il controllo è gestito in maniera sincronizzata, mediante messaggi asincroni, dai due pezzi del middleware, che si mettono d'accordo sul quando restituire il controllo ai due programmi. Avendo due programmi che hanno entrambi il controllo e che devono comunicare tra loro metto un pezzo, diviso in due oggetti concorrenti, in mezzo ai due che decidono quando dare il controllo ad uno e quando darlo all'altro, in modo tale da farli avanzare in maniera asincrona. Il wrapper si connette quindi al client:

- **input:** il wrapper accetta richieste da parte del client che usa il LIS, le ristruttura poichè devono essere in una forma che sia utilizzabile da parte del LIS e invoca i target object con i componenti strutturati
- **output:** il wrapper cattura gli output dell'oggetto remoto e li confeziona in modo tale che possano essere usati dal client.

Molto spesso il meccanismo di comunicazione è complesso ed articolato.



19.5 Tipi di wrapping

I wrapper sono suddivisi in 4 categorie: Database wrappers, System service wrappers, Application wrappers e Function wrappers.

Coscienze
Associazione

19.5.1 Database wrappers

Esso può essere a sua volta classificato in:

- **forward wrapper:** Avendo un componente nuovo che deve accedere direttamente ai dati gestiti dal LIS, mediante forward wrapper è possibile accedere a tali dati
- **backward wrapper:** serve quando è la legacy application che deve accedere ai dati del sistema, ovvero quando è il LIS che deve accedere ai dati salvati sul nuovo sistema

19.5.2 System service wrapper

Si ha un LIS che fornisce servizio e dovendo utilizzare tale servizio. Si ha quindi un adapter che mi permette di usare il servizio del LIS

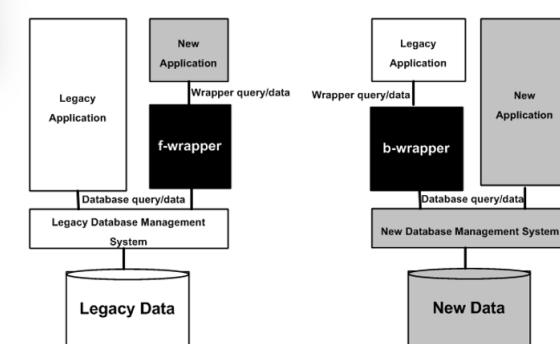


Figure 5.1Forward wrapper © ACM, 2006

Figure 5.2 Backward wrapper © ACM, 2006

Figura 19.1: Database Wrapper

19.5.3 Application wrapper

Si va a **wrappare un intero processo batch**, dove un processo batch potrebbe essere ad esempio la lettura da parte di una banca di una lista di utenti. **Questo wrapper permette di includere legacy components come oggetti**. Questi oggetti verranno poi invocati per produrre report o files aggiornati. Questo in quanto i processi batch non hanno un'interfaccia, ma vengono lanciati a linea di comando. Esistono anche **on-line transactions**. Devo quindi **wrappare un'intera transazione on line**. Ho quindi un **programma interattivo e lo incapsulo a livello di interfaccia utente**. Sono spesso gli on-line transaction models che vanno a gestire i programmi interattivi.

19.5.4 Function wrapper

Questo wrapper **fornisce un'interfaccia per chiamare funzioni all'interno di una wrapped entity**. In questo meccanismo, **solo una parte del programma e non l'intero programma, è invocato**. È quindi necessario usare tecniche **per wrappare il programma in più andando a modificarlo per garantirmi l'accesso solo a quella parte del programma senza però avere errori**. Per usare questa tecnica devo quindi anche **modificare il programma LIS**. Se il programma non è un **programma batch**, anche questo wrapping risulta più complesso.

19.6 Livelli di incapsulamento

Sneed ha classificato **5 livelli di granularità**. Le **procedure sono a livello più basso mentre i processi sono a livello più alto**.

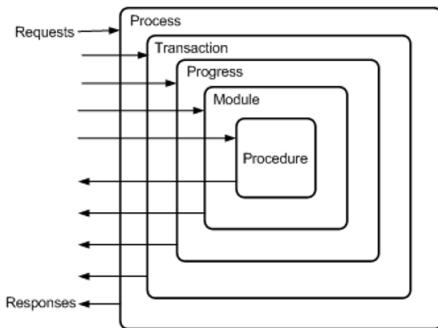


Figure 5.3 Levels of encapsulation ©IEEE, 1996

Figura 19.2: Livelli di incapsulamento

19.6.1 Process level

Questo livello fa riferimento all'applicazione batch. Input ed output sono scritti in un file con cui il programma batch comunica. Il wrapper deve passare gli input ad un file che sono richiesti in input al processo. Quest'ultimo produce degli output che sono poi presi dal wrapper e che vengono passati all'applicazione client.

19.6.2 Transaction level

Il wrapper va a simulare uno user terminal. Tale livello richiede la modifica del codice del programma legacy per fare in modo di rigirare tutta la I/O verso il wrapper invece che verso il terminale

19.6.3 Program level

Devo avere delle API che permettono al wrapper di comunicare con il programma, di invocarlo e di passare i parametri. Nel caso in cui però il formato dei dati sia differente dal formato che il LIS vuole, bisogna prendere i dati e convertirli. In aggiunta l'output del programma sono presi dal wrapper e riformattati, facendoli tornare al loro tipo originale. Sono poi inviati al client

19.6.4 Module level

Si usano le interfacce standard. Qui il concetto è che il programma client ed il modulo si trovano in spazi di indirizzamento diversi. In tal caso bisogna passare l'oggetto non per riferimento ma per valore. Come prima cosa quindi il wrapper bufferizza i valori ricevuti nel proprio address space. Successivamente tali valori sono passati al modulo invocato. L'output del metodo invocato viene poi passato al cliente

19.6.5 Procedure level

Concetto di function visto prima. Io sto invocando una parte del programma, dovendo ricostruire l'interfaccia del programma e dovendo definire i dati di cui quella parte del programma ha bisogno. è quindi necessario ricostruire l'interfaccia utente del programma. è come se volessi estrarre un pezzo di codice dal programma LIS ed incapsularlo in un altro programma. Facendo ciò, devo fare però data flow analysis per ricostruire l'interfaccia di quel pezzo di codice, definendo input ed output per quel pezzo di codice. Possibili variabili di input sono variabili usate in quel pezzo di codice ma non definite al suo interno. Variabili di output potrebbero essere invece variabili definite che però non vengono usate in quel pezzo di codice.

19.7 Costruire un wrapper

Un LIS è incapsulato in 3 passaggi:

- un wrapper è costruito
- si adatta il target program
- si verificano le interazioni tra target program e wrapper

Il wrapper prende quindi input dal client, li trasforma in input validi per il target program e li invia. Intercetta poi gli output, ritrasformando i dati e li invia al client. Se i linguaggi di programmazione sono diversi, realizzare un wrapper potrebbe risultare complesso.

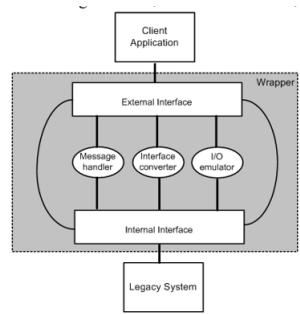


Figure 5.4 Modules of a wrapping framework

Figura 19.3: Costruire un wrapper

Si necessiterà di un'interfaccia esterna che sarà quella che utilizza la client application, mentre l'interfaccia interna userà il sistema legacy. Il sistema legacy in questo caso invoca l'internal interface, ovvero solo una delle due interfacce che il wrapper ha. Nel caso dell'adapter è il wrapper che invoca il sistema legacy. Dovendo realizzare module level, program

level, process level, sarà sempre il wrapper ad invocare l'oggetto legacy. Viceversa, nel transaction level è il LIS che invoca l'interfaccia interna del wrapper. Si ha poi un message handler, un'interface converter ed un I/O emulator. Tipicamente le interfacce comunicano mediante il message handler.

19.7.1 External interface

Si ha quindi un meccanismo al client che gli permette di accedere all'interfaccia del wrapper. Se il wrapper è scritto in un linguaggio diverso, rispetto al programma client, si può aggiungere un altro strato che è scritto nello stesso linguaggio del programma client, in modo tale da nascondere la tecnologia con cui è realizzato il wrapper. Sarà poi questo strato a comunicare con l'interfaccia wrapper. Si possono costruire più strati, in base a quanto voglio fare information hiding. Fondamentalmente metto un adapter che invoca il wrapper, nascondendo quindi al programma client il modo in cui è realizzato il wrapper, in quanto l'adapter è scritto allo stesso modo del programma client. I messaggi che arrivano sono composti da un header e da un corpo, dove l'header contiene informazioni di controllo come: l'identità del sender, timestamp, transaction code, tipo di target program, l'identità della transazione, procedura, programma o metodo invocato. Il corpo contiene gli argomenti di input. Solitamente il contenuto del messaggio sono stringhe, delimitate da uno slash. Il wrapper deve anche fungere da gestore della concorrenza in quanto un LIS potrebbe essere invocato da più programmi, dovendo quindi gestire le chiamate di diversi client.

19.7.2 Internal interface

Interfaccia visibile al server. Essa dipende dal linguaggio e dal tipo della wrapped entity. Ci sono diversi casi che possono verificarsi:

- l'entity è un job: in questo caso il wrapper è una job control procedure
- se invece è una transaction, un programma, procedura o modulo, l'interfaccia interna conterrà la lista di parametri del software encapsulato

Tipicamente, se il wrapper può invocare direttamente l'oggetto server per prendere i risultati, la chiamata non avviene dal componente LIS al wrapper ma al contrario. Se quindi il LIS offre un servizio, sarà il servizio ad essere invocato.

19.7.3 Message handler

Essendoci concorrenza, **devo serializzare**. Per **serializzare** si ha quindi un **message handler**. Esso **bufferizza messaggi di input ed output**. Se si ha invece un componente che **gestisce già la concorrenza** il **message handler** non serve più a tale scopo. Dovendo **wrappare un qualcosa che si trova su pc**, si potrebbe usare una **driving cleaning library**, ovvero oggetto che gestisce la concorrenza. Invocando quindi un oggetto, quest'ultimo viene invocato mediante meccanismo che non gestisce la concorrenza, è necessario inserire il **message handler** per gestirla.

19.7.4 Interface converter

Parte del componente che si occupa di convertire i parametri dal client al LIS e viceversa.

19.7.5 I/O Emulator

Intercetta gli **input** e gli **output** e riempie i buffer di **input**. L'I/O Emulator serve soprattutto quando ho l'applicazione che deve essere usata a livello di user interface. Simulo quindi l'I/O del programma legacy.

19.8 Adattare un programma al wrapper

Posso inoltre **adattare il programma che devo wrappare**, in modo tale che il programma possa continuare a funzionare normalmente. I programmi sono adattati con tools, raramente a mano. I tipi di tool raccomandati sono: **transaction wrapper**, **program wrapper**, **module wrapper** e **procedure wrapper**.

19.9 Screen scraping

Altro modo per **definire il wrapping di un programma**, passando da **text-based interfaces a graphical interfaces**. La differenza con il **transaction wrapping** è che in quel caso c'è un **transaction processing monitor**, ovvero coloro che si occupano di gestire i programmi interattivi. Si continua ad usare quindi un **LIS** mediante **user interface**. L'idea è di estrarre mediante **reverse engineering** il **modello dell'interfaccia utente**, analizzando tutte le screen sections. Si creava poi una nuova **GUI** che poteva essere **reengineerizzato in una web app**.

Nello screen scraping invece non si hanno i transaction processing monitor ma il meccanismo per wrappare a livello di user interface è lo stesso. L'unica cosa è che non mi devo sostituire al transaction processing monitor ma mi devo sostituire alla gestione di I/O che viene fatta su pc. In entrambi i casi però si sta wrappando un sistema interattivo, ovvero un sistema che ha I/O. Screen scraping è una soluzione a breve termine per problemi più grandi. Problemi legati allo screen scraping, ovvero solo all'aggiunta di una GUI ad un LIS sono:

- non si evolve e non supporta nuove funzionalità
- i costi di mantenimento sono elevati
- ignora i problemi di overloading (abilità di una funzione di eseguire task differenti)

19.10 Migration

Molto spesso il wrapping viene usato quando si vuole fare migration. Ho quindi un software che non evolve, per cui si potrebbe inserire il sistema all'interno di un altro software. Per fare ciò però è necessario incapsularlo, mantenendo le funzionalità. Per migrare devo renginerizzare, ma si sta migrando però verso una nuova tecnologia, ovvero si sta sostituendo il sistema scritto con una specifica tecnologia con un sistema scritto in una tecnologia differente. Posso quindi migrare il database, migrando i dati che esso contiene. È possibile inoltre migrare i programmi. Se tutto ciò afferisce allo stesso sistema, si parla di approccio incrementale, poiché viene migrato un sistema non decomponibile ma poco alla volta, migrando l'interfaccia utente per consentire agli utenti di accedere mediante web, poi si può migrare il database. In tal modo, se si vogliono costruire componenti nuove è possibile accedere al database in locale. Si migrano poi le componenti, definendo componenti nuove che vanno a sostituire i componenti legacy. La migrazione comprende 3 passi principali:

- Estrarre i dati, trasformarli nel caso ci siano tipi differenti tra nuovo sistema e sistema LIS e successivamente devo caricarli.
- Potrei inoltre dover eseguire della data cleaning. Non avendo vincoli di integrità nel LIS, potrei avere dei dati sporchi, per cui essi devono essere puliti da problemi che magari, i vincoli del nuovo sistema mi consentono di rilevare

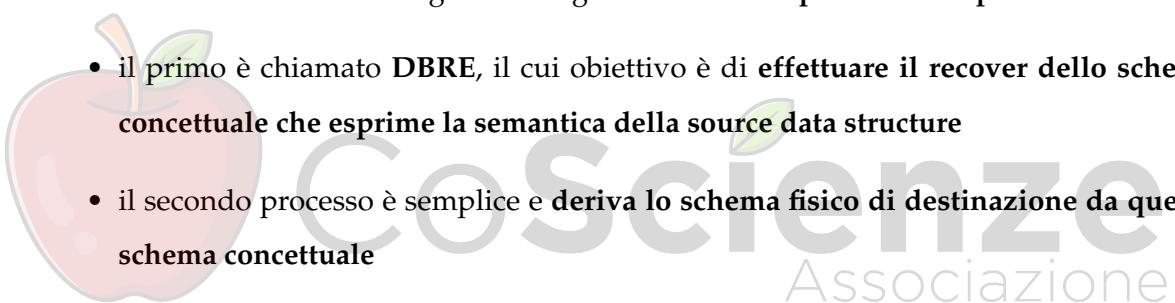
- Con la **program conversion**, devo applicare un approccio semi-automatico, intervenendo per scrivere del codice nella nuova tecnologia che sia coerente con la tecnologia, e non basato sulla tecnologia vecchia. Se io faccio **migrazione**, è perchè voglio aggiungere funzionalità e voglio evolvere il sistema, cosa impraticabile con il sistema legacy impraticabile nella direzione in cui voglio

Se migro completamente il sistema legacy, posso usarlo come sistema per verificare il nuovo sistema quando faccio regression testing. Posso quindi usare l'oracolo che viene dal LIS.

19.11 Schema conversion

Per **schema conversion** si intende la conversione dello schema del database legacy in una struttura di database equivalente espressa nella nuova tecnologia. La trasformazione dal database dello schema sorgente al target schema è composto da due processi:

- il primo è chiamato **DBRE**, il cui obiettivo è di effettuare il recover dello schema concettuale che esprime la semantica della source data structure
- il secondo processo è semplice e deriva lo schema fisico di destinazione da questo schema concettuale



19.12 Data conversion

Per **conversione dei dati** si intende lo spostamento delle istanze di dati dal database legacy al database di destinazione. La conversione dei dati richiede tre passaggi: estrazione, trasformazione e caricamento (ETL). Innanzitutto, si estraggono i dati dall'archivio legacy. In secondo luogo, si trasformano i dati estratti in modo che le loro strutture corrispondano al formato. Inoltre, si esegue la pulizia dei dati (ovvero scrubbing o pulizia) per correggere o eliminare i dati che non si adattano al database di destinazione. Infine, si trasportano i dati trasformati nel database di destinazione.

19.13 Program conversion

Nel contesto della **migrazione LIS**, per conversione di programma si intende la modifica di un programma per accedere al database migrato anziché ai dati legacy. Il processo

di conversione lascia invariate le funzionalità del programma. La conversione del programma dipende dalle regole utilizzate per trasformare lo schema legacy nello schema di destinazione.

19.14 Coerenza di LIS e sistema target

È importante garantire che i risultati del sistema target siano coerenti con quelli del LIS. Pertanto, non è necessario introdurre nuove funzionalità nel sistema di destinazione in un progetto di migrazione. Se sia il LIS che il sistema di destinazione hanno la stessa funzionalità, è più facile verificarne la conformità. Tuttavia, per giustificare la spesa del progetto e il rischio, in pratica, i progetti di migrazione spesso aggiungono nuove funzionalità.

19.15 Cut over e Roll over

L'eventualità di passare al nuovo sistema da quello vecchio è necessario per causare interruzioni minime al processo aziendale. Esistono tre tipi di strategie di transizione:

- **cut-and-run:** spengo il sistema legacy e accendo quello nuovo. Tale tecnica però è molto pericolosa per cui si preferiscono usare tecniche incrementali
- **phased interoperability:** procedo in modo incrementale spegnendo una funzione nel legacy e accendendola nel nuovo sistema. Metto alla fine in funzione il nuovo sistema facendo spegnere il vecchio
- **parallel operation:** mantengo entrambi i sistemi in esercizio in modo da verificare che il nuovo funzioni prima di spegnere il vecchio.

19.16 Migration planning

La migrazione va pianificata nel dettaglio. Il processo di migrazione è quindi molto complesso. Prima di decidere se procedere o meno è necessario prendere in considerazione diversi fattori. La migrazione infatti è solo una delle possibili alternative. Seguono di seguito i vari passaggi

19.16.1 Perform portfolio analysis

Quando faccio l'**analisi di un sistema per decidere se migrarlo o no**, devo prendere in considerazione due aspetti: **qualità tecnica del sistema** e la **business value**. La **qualità tecnica** è misurata in base al:

- livello di **obsolescenza tecnologia**
- quanto è **decomponibile il sistema**, quanto è modularizzabile
- **architettura del sistema**

Anche il **valore aziendale** però ha un suo peso. Oltre al livello di obsolescenza della tecnologia, gli altri aspetti possono degradarsi a causa dell'evoluzione del sistema. Facendo evoluzione del sistema molto spesso, la qualità tecnica potrebbe diminuire. Per il valore aziendale invece, si fa riferimento a quanto il software fa quello che deve fare per l'azienda. Un sistema quindi **non evolve e perde business value**. Nel caso in cui abbia un **basso valore di business**, il sistema non posso nemmeno utilizzarlo per fare redevelopment. Essendo disallineato, ciò che fa il sistema è inutile. Per questo che, nel portfolio è definito un grafo, diviso in quattro quadranti:

- **quadrante 1: bassa qualità e basso valore di business**
- **quadrante 2: alta qualità e basso valore.** Il sistema non è evoluto come doveva. Alcune parti del sistema non sono allineate. Tali componenti possono però essere rimpiazzati con COTS, ovvero il sistema ha qualcosa di utile e qualcosa di non utile. Avendo una buona architettura è possibile rimpiazzare solo alcune delle sue componenti disallineate. Si potrebbe in alternativa fare manutenzione per riallineare le componenti ai requisiti. Avendo un buona architettura posso fare migrazione o sostituzione delle componenti obsolete. Se il sistema è del tutto obsoleto però non posso fare nulla per migliorare l'utilità
- **quadrante 3: alta qualità ed alto valore.** Idealmente immagino che un sistema sia all'inizio nel quadrante 3. Finché si sta in questo quadrante, il sistema sta evolvendo ma su sta anche facendo operazioni di ristrutturazione e renginerizzazione per mantenere la qualità del sistema alta.
- **quadrante 4: bassa qualità ed alto valore.** Il sistema è evoluto come doveva ma non sono state fatte operazioni di ristrutturazione per migliorarne la qualità. è un buon candidato per la migrazione

I tipi di **reengineering** e di migrazione che vado a fare dipendono da vari casi. Quando si ha un'alta obsolescenza tecnologia, il software però è molto propenso alla migrazione. Il software è buono ma a causa del linguaggio è complesso da far evolvere. Si può passare da quadrante 3 o al quadrante 4 o al quadrante 2, e da questi due si può poi passare al quadrante 1.

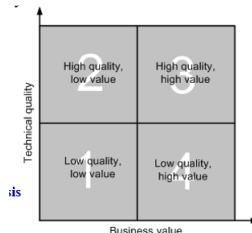


Figura 19.4: Perform portfolio analysis

19.16.2 Identificare gli stakeholders

Gli stakeholders sono le persone dell'organizzazione che influenzano il comportamento del sistema. Essi includono: architetti, developers, mantainers, managers, customers ed users. Essi andranno poi a valutare l'impatto ed il risultato del migration project. I requisiti sono descrizioni di ciò che gli stakeholders vogliono che il sistema faccia. Ci sono due sfide nella definizione dei requisiti.:

- Innanzitutto, assicurati che vengano acquisiti i requisiti giusti.
- In secondo luogo, esprimere i requisiti in modo tale che le parti interessate possano facilmente rivedere e confermare la loro correttezza.

19.16.3 Creare un business case

Sulla base di un **business case**, la direzione esecutiva può decidere se il progetto di migrazione aumenterà o meno la qualità, ridurrà i costi di manutenzione e sarà finanziariamente sostenibile. In generale, un **buon business case** fornisce le seguenti informazioni sul progetto di migrazione: Dichiarazione del problema, Soluzione, Rischi e vantaggi.

19.16.4 Go-on o no-go decisions

Una volta che un **business case** è stato definito, viene esaminato dalle parti interessate per raggiungere un accordo. Se il **business case** non è soddisfacente, il progetto di migrazione legacy viene terminato in questo passaggio.

19.16.5 Comprendere il LIS

La **comprendere del LIS** è essenziale per il successo di qualsiasi progetto di migrazione. Le **tecniche disponibili** per affrontare questa sfida includono la **comprendere del programma e il reverse engineering**.

19.16.6 Comprendere la target technology

Questa attività può **procedere in parallelo** con l'attività di comprendere del LIS. È importante **comprendere le tecnologie che possono essere utilizzate nell'attività di migrazione e le tecnologie che sono state utilizzate nel sistema legacy**. In generale, quattro tipi di tecnologie sono di interesse nello sforzo di migrazione. È necessario confrontare e **confrontare tutte le tecnologie disponibili per valutarne le capacità**. Se le funzionalità si sovrappongono, dobbiamo valutare queste tecnologie per **comprendere la qualità del servizio che forniranno nel processo di migrazione**. Per formulare l'eventuale architettura e progettazione del sistema, vengono eseguite tali valutazioni.

19.16.7 Definire la target architecture

L'architettura di destinazione è l'architettura desiderata del nuovo sistema. Essa **model-
la la visione degli stackholder del nuovo sistema**. Questo di solito richiede descrizioni che utilizzano viste diverse con diversi livelli di granularità. È probabile che l'architettura di destinazione **si evolva durante il processo di migrazione**. Pertanto, l'architettura di destinazione viene continuamente rivalutata e aggiornata durante il processo di migrazione.

19.16.8 Definire la strategia

Una **strategia definisce il processo complessivo di trasformazione del LIS nel nuovo sistema**. Ciò include la **metodologia di migrazione**, ovvero la **conversione dello schema, la conversione dei dati e la conversione del programma, il test e il cut over**. Per un **sistema legacy mission-critical, distribuire il nuovo sistema tutto in una volta è una procedura rischiosa**, quindi un sistema legacy viene evoluto in modo incrementale al nuovo sistema. Durante l'attività di migrazione, molte cose possono cambiare: **i requisiti degli utenti possono cambiare, possono essere acquisite ulteriori conoscenze sul sistema e la tecnologia può cambiare**. Tali cambiamenti devono essere adattati allo sforzo migratorio. Pur adattando tali cambiamenti, **una strategia di migrazione deve ridurre al minimo i rischi, ridurre al**

minimo i costi di sviluppo e distribuzione, supportare una pianificazione aggressiva ma affidabile e soddisfare le aspettative di qualità del sistema.

19.16.9 Riconciliare la tecnologia con le necessità degli stakeholders

È necessario sviluppare un consenso tra le parti interessate prima di attuare il piano di migrazione. La strategia di migrazione sviluppata nella fase precedente deve essere conciliata con le esigenze delle parti interessate. Pertanto, questo passaggio include informare le parti interessate sull'approccio, rivedere l'architettura di destinazione e la strategia. L'intero gruppo valuta la strategia e fornisce input per il profilo di consenso finale.

19.16.10 Determinare le risorse necessarie

Stimiamo il fabbisogno di risorse, compresi i costi di implementazione del progetto. Si può utilizzare il modello di stima dei costi ampiamente utilizzato chiamato Constructive Cost Model II (COCOMO II). COCOMO II affronta i modelli di processo non sequenziali, il lavoro di reingegnerizzazione e l'approccio basato sul riutilizzo. Il modello COCOMO II fornisce stime dello sforzo, pianificazione per fasi e personale per fasi e attività.

19.16.11 Valutare la fattibilità della strategia

Dopo aver eseguito i primi 12 passaggi, la gestione deve avere una conoscenza del sistema sottoposto a migrazione, delle opzioni tecnologiche disponibili, dell'architettura di destinazione, della strategia di migrazione, dei costi della migrazione e di una pianificazione per effettuare la migrazione. Sulla base delle informazioni disponibili, la direzione determina se la strategia di migrazione è fattibile o meno. Se la strategia risulta praticabile, il piano di migrazione viene finalizzato. D'altro canto, se è inaccettabile, viene prodotta una relazione dettagliata. Sulla base delle ragioni esposte nella relazione, è possibile rivedere la strategia migratoria fino a quando:

- può essere individuato un approccio fattibile, o
- la strategia di migrazione è ritenuta irrealizzabile e il progetto è terminato.

19.17 Metodi di migrazione

Esistono diversi metodi alla migrazione. Di seguito ne sono riportati 7, di cui non tutti sono compatibili.

19.17.1 Cold turkey

Strategia che si riferisce all'approccio **big bang**. Non incrementale è come se facessi renginerizzazione e poi implementassi tutto il sistema. Tale approccio è molto rischioso e del tutto sconsigliato. Il rischio del fallimento cresce con la grandezza del sistema che deve essere migrato. Può essere adottato se uso il LIS ad una funzionalità stabile, per cui il sistema non viene evoluto ed è di piccole dimensioni. Stabile ovvero **non deve evolvere troppo intanto che faccio migrazione**.

19.17.2 Database first

Approccio in cui **vado a migrare prima il database**. Fatto ciò **vado a vedere di cosa ho bisogno**. Si potrebbe necessitare un **forward gateway**, similare al **backward wrapper**. Il concetto è lo stesso ma **in questo caso il contesto è di migrazione mentre prima era un contesto di wrapping**. Migrando prima il db si ha bisogno di un gateway che mi consenta di accedere al database. Se il LIS vuole accedere al database, è quindi necessario un gateway.

Si migra prima il database in quanto si ha bisogno di sviluppare nuove componenti che accedono al database. Ho bisogno di un **forward gateway** per consentire al sistema legacy di accedere al target database. Tutto ciò viene fatto mediante un **processo di database reverse engineering e successiva renginerizzazione del database**. Successivamente alla migrazione del db, alla creazione di nuove componenti, **mi preoccupo di migrare la parte vecchia**.

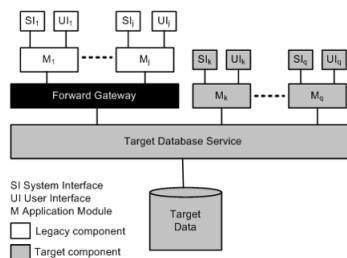


Figura 19.5: Database first

Per poter essere applicabile devo accedere al database. Alcuni **problemi che possono verificarsi sono**:

- Per poter accedere al database, lo **strato della gestione dei dati deve essere separato dallo strato di application logic**, altrimenti diventa complicato.
- La **struttura del database dovrà essere inoltre definita prima dell'inizio della migrazione**. Devo prima definire la struttura e devo poi realizzare la migrazione dei dati.

- Può essere difficile realizzare un gateway, in quanto può essere complesso capire il contesto del database

19.17.3 Database last

è applicabile solo in caso di LIS decomponibili. In questo caso i LIS sono migrati incrementalmente alla target platform e la migrazione del database è fatta per ultima. Si avrà un **reverse gateway**, in quanto si consentirà alla parte nuova di accedere ai legacy data. Similmente al first approach, l'information system è supportato dai gateway. I due principali problemi con questo approccio sono:

- mentre nel database first io ho migrato lo schema per cui il forward gateway deve preoccuparsi del mapping, in questo caso il nuovo schema deve essere ancora costruito, in quanto il nuovo sistema deve accedere ai dati secondo il nuovo schema di dati ed il reverse gateway deve mappare il nuovo schema sul vecchio schema.
- In secondo luogo, potrebbero esserci delle funzionalità del nuovo dbms che non possono essere sfruttate, in quanto quest'ultimo non è ancora presente.

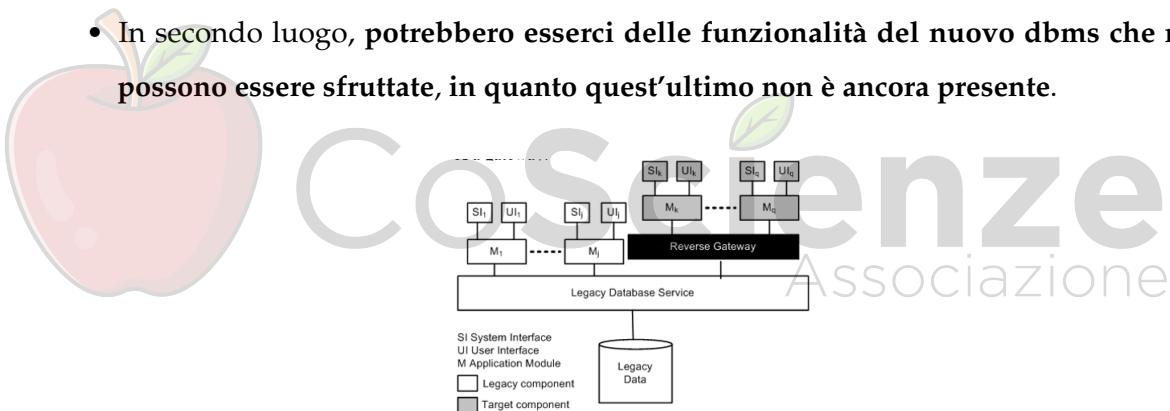


Figura 19.6: Database last

19.17.4 Composite database

La migrazione di dati procede in parallelo con la migrazione del sistema. Immaginando di dover dividere il sistema in sottosistemi. Ogni sottosistema accede principalmente a dei dati, per cui si cerca di accoppiare ad ogni sottosistema, parte di database che è più importante per quella parte, procedendo con la migrazione di quei dati e di quel sottosistema. Potrebbe capitare però che il sistema debba accedere ai dati sulla parte vecchia, per cui si utilizzerà un coordinatore per gestire sia il reverse gateway che il forward gateway. Il mapping dovrà essere fatto solo su una parte dei dati e non su tutti i dati. La migrazione dei dati sarà anch'essa incrementale. Ci può essere un overlap tra dati nella parte vecchia e

dati nella parte nuova. In questo processo di migrazione incrementale, è possibile avere dei dati duplicati in entrambi i database. Ciò dipende dal sistema che sto considerando. Si avrà una mapping table per indicare dove i dati siano e se essi siano duplicati.

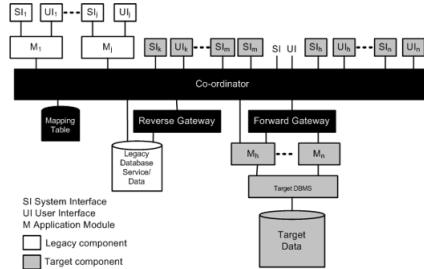


Figura 19.7: Composite database

19.17.5 Chicken little

Approccio incrementale che raffina la tecnica di composite database. Anch'esso usa entrambi i database, target e legacy. Tale tecnica usa l'application gateway che consente di accedere ai dati legacy e ai dati nuovi.

Questo va bene per decomposable systems in cui i dati sono separati dal resto dell'applicazione. Se il sistema è però non decomponibile, il gateway si presenta tra users ed external information systems. Il gateway sarà realizzato come un wrapper ma sarà pensato soprattutto per wrappare il servizio di database. Non si avrà un'interfaccia verso l'interno, andando a convertire semplicemente le query. Il gateway è quindi un wrapper particolare che serve a wrappare il database.

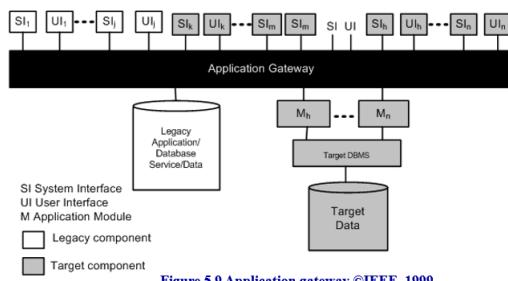


Figure 5.9 Application gateway ©IEEE, 1999

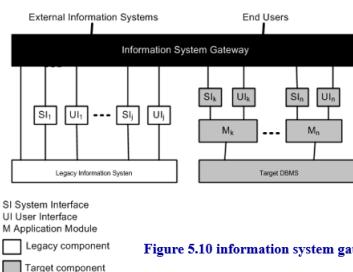


Figure 5.10 information system gateway ©IEEE, 1999

Figura 19.8: Chicken little

Parlando di gateway a livello di sistema, si sta wrappando l'intera applicazione. Vado a migrare la nuova applicazione, potendo accedere direttamente alla parte nuova e tramite wrapper alla parte vecchia. Anche per questa tecnica ci sono degli steps precisi da seguire. Fondamentalmente, quando si migra il sistema legacy è necessario considerare il grado

di decomponibilità. Se il sistema è **non decomponibile**, bisogna partire con la migrazione dell’interfaccia utente, effettuando uno **stream scraping** e permettendo di rendere il sistema **utilizzabile nel nuovo contesto**. Si migra poi il db in modo unico o incrementale. Avendo già l’interfaccia infatti sarà più semplice la migrazione. Chciken little usa l’approccio **composite database** per cui migra incrementalmente. Nel caso di sistemi non decomponibili è quindi sempre necessario fare un **wrapping** a livello di **user interface**. Non posso fare altro. Se il sistema invece è decomponibile, migro a pezzi tutto, anche l’interfaccia utente. Seppur il sistema fosse del tutto decomponibile però, non so se mi conviene migrare a pezzi. È sempre meglio migrare prima l’interfaccia utente per tutte le funzionalità che devono essere migrate. Se io ho un utente che deve usare certe funzionalità non posso far usare alcune componenti in un modo e altre componenti in un altro. Seppur decomponibile conviene quindi migrare l’interfaccia utente in una volta, andando poi a migrare a pezzi il sistema. L’interfaccia per la parte nuova o vecchia sarà quindi la stessa

19.17.6 Butterfly

Metodologia incrementale per migrare i dati. Si è definito il database e una volta fatto questo devo migrare i dati. Per migrare i dati dal sistema vecchio al sistema nuovo, prima di accendere il sistema nuovo, potrebbe richiedere giorni. Posso quindi permettermi di avere il sistema spento per qualche giorno? Devo necessariamente spegnerlo altrimenti intanto che migro i dati essi cambiano e accedendo il sistema nuovo, i dati saranno obsoleti.

Butterfly quindi propone una strategia incrementale per migrare i dati intanto che il sistema è in esercizio. Esso propone dei temporary store (TS), in cui c’è la parte di dati che deve essere migrata. Il sistema concede l’accesso mediante dei gateway sia alla parte vecchia che alla parte nuova. Similare al composite però in quel caso era lo schema ad essere migrato incrementalmente. Per cui si usa una sequenza di TS. **Decido quale sia il pezzo che deve essere migrato.** Lo metto in un TS e lo inserisco poi nell’altra parte, tenendo consistenti i dati. Il TS deve essere sufficientemente piccolo da consentirmi di effettuare questa cosa. Durante la migrazione quei dati non saranno accessibili per cui questa operazione viene fatta quando si può spegnere il sistema, o se non posso spegnere il sistema i dati non devono essere toccati. Per migrare i dati è quindi necessaria una strategia incrementale. Il tempo in cui i dati non devono essere accessibili dovrà essere molto piccolo.

CAPITOLO 20

Metriche del software

20.1 Introduzione: Misure ed ingegneria del software

Le misure rappresentano un aspetto cruciale in tutte le branche dell'ingegneria... tranne che nell'ingegneria del software. Nell'ingegneria del software si parla di proprietà come usabilità, affidabilità, e manutenibilità senza spiegare come possano essere misurate. Devo però poter misurare per decidere se requisiti non funzionali sono soddisfatti e se il sistema ha certe caratteristiche di qualità, da qui la frase "you cannot control what you cannot measure". È necessario però definire cosa misurare. Si può misurare non solo il prodotto, ma anche attributi di processo come il costo di sviluppo, la produttività, la difettosità, l'affidabilità...

Il fatto di non avere un approccio manageriale al processo di sviluppo del software fu uno dei motivi che causò la crisi del software. Si arrivò ad un punto in cui non si riusciva a soddisfare nuove esigenze, sviluppando software nuovo, in quanto tutto lo sforzo era dovuto alla manutenzione degli esistenti.

Il modello FURPS è un modello basato su: Functionality, usability, reliability, performance e supportability (riguarda manteinability e portability). Esso è un modello di qualità del prodotto software. Esse sono le caratteristiche rispetto i quali si esprimono i requisiti non funzionali. Alcune caratteristiche però possono essere espresse in termini di altre come availability, affidabilità... che posso comunque rispettare. Esse però sono caratteristiche esterne, ovvero caratteristiche visibili o percepibili dall'utente. Si parla invece di caratteri-

stiche interne se si parla di manutenibilità. Di solito quello che si fa è **definire un modello di qualità come ISO 9000 di tipo gerarchico, in cui si parte dalle caratteristiche esterne, si individuano le sottocaratteristiche interne, e tutte esse vengono poi mappate.**

Esempio: sulla base del numero di McCabe posso avere un'idea della testability del programma. Un programma è testabile se riesco a derivare facilmente casi di test. Riesco inoltre a testare in maniera sistematica il sistema. Se però si hanno molti cammini diventa comunque difficile testarlo. Un programma con gli if in cascata è molto più semplice da testare rispetto ad if in serie. Testability però è una caratteristica interna. Il numero di cammini è un indicatore della testability. Il fatto che però sia robusto è una caratteristica esterna. Posso vedere quanti sono le parti di codice che possono sollevare eccezioni e quante sono le eccezioni gestite, ciò viene fatto vedendo il codice. Ciò non è una misura di robustezza ma può essere un indicatore.

20.2 Modelli di qualità

I modelli di qualità sono usati per definire indicatori (misure interne) che possono essere dei buoni predittori per le misure di caratteristiche esterne. Pur essendo che i requisiti non funzionali sono definiti sulla base di caratteristiche esterne, si vanno a misurare le caratteristiche interne per capire se rispetto le caratteristiche esterne.

I design goal che ricaviamo dai requisiti non funzionali, rappresentano gli obiettivi di progettazione. Alcuni obiettivi potrebbero essere le performances, per cui si cerca di scrivere codice veloce, o obiettivi di manutenibilità, per cui si cerca di modularizzare il sistema. Quando progetto, lavoro sulle caratteristiche interne del prodotto e misuro le caratteristiche interne, sapendo che le caratteristiche interne avranno un impatto sulle caratteristiche esterne. Le caratteristiche interne sono statiche, per cui non devono essere testate. Si ha una validazione empirica rispetto a quale caratteristica interna influisce sulla caratteristica esterna.

20.3 Teoria della rappresentazione della misura

Fornisce un quadro di riferimento rigoroso che permette di valutare la bontà di una misura. Esso definisce alcuni concetti come misura, scala e trasformazione di scala. Sulla base di tale teoria è possibile analizzare le misure proposte in letteratura e proporne di

nuove/migliori. Quello che devo fare è legare le misure a fenomeni empirici, ovvero deve esserci una validazione empirica di quella misura.

Esempio: come faccio a dire che la scala Celsius è una buona scala? Quello che è stato fatto è definire a 0 il congelamento e 100 ebollizione, avendo una scala tra i due valori.

20.4 Misure

La misura è un **processo di assegnazione di un simbolo ad attributi o proprietà di un oggetto per poterlo descrivere secondo regole ben definite.** Una definizione alternativa è che **una misura è l'attribuzione di un numero (o di un simbolo) ad un attributo di una entità per caratterizzarlo.** La **misura non è un numero ma una relazione tra attributi e numeri.** Per effettuare una misura dobbiamo avere una chiara idea di quali attributi vogliamo misurare e di quali entità possiedano tale attributo. La comprensione empirica precede la misura. Se io dico che il **prezzo di un prodotto è x, sto assegnano un simbolo**, un valore, **ad un attributo di quel prodotto** che è il prezzo. le misure vengono usate in fisica, ingegneria, economia... Ciò che hanno in comune le misure, è il **processo per ottenerle.**

20.5 Misure dirette ed indirette

Una **misura diretta di un attributo** è qualcosa che posso misurare direttamente. Altrimenti si parla di **misura indirette**. Per le **misure indirette**, la **misurazione di un attributo avviene mediante la misurazione di un altro attributo.**

Esempio: il numero di chiamate per eseguire un caso d'uso è una misura diretta. L'altezza è una misura diretta. La misura con termometro a mercurio è indiretta in quanto si sta misurando l'allungamento della barretta di mercurio che però è graduata in modo tale da indicare la temperatura

20.6 Sistema di relazioni empiriche

Avere una **comprensione empirica** significa avere un **insieme di relazioni (Re)** che legano tra loro le entità (E) in esame. **Esse sono relazioni che osservo.**

Esempio: Entità: personaggi dei fumetti. Attributo: altezza. Relazioni: Re1(Pippo,Topolino) se “Pippo è più alto di Topolino”, Re2(Pippo) se “Pippo è alto”, Re3(Topolino, Pippo, Orazio) se “Topolino è più alto di Pippo quando siede sulle spalle di Orazio”

20.7 Sistema di relazioni numeriche

Un sistema di relazioni numeriche è composto da un insieme di simboli (N) e da un insieme di relazioni (R_n) che legano tra loro tali simboli. Vogliamo esprimere in termini di relazioni numeriche ciò che osserviamo in termini di relazioni empiriche. Quello che possiamo fare è tradurre un sistema di relazioni empiriche in un sistema di relazioni numeriche.

Esempio: Insieme numerico: Numeri reali. Relazioni: $Rn1(x,y)$ se $x > y$, $Rn2(x)$ se $x > 180$, $Rn3(x,y,z)$ se $0.7x + 0.8z > y$

20.8 Relazioni tra sistemi di relazione

Dato un sistema di relazioni empiriche e un sistema di relazioni numeriche si definisce “misura” una relazione M tra i due sistemi tale che:

- ad ogni entità è associato un numero
- ad ogni relazione tra entità corrisponde una relazione tra numeri
- l'esistenza di una relazione tra istanze di entità implica l'esistenza della corrispondente relazione tra i numeri(misure) associate a quelle istanze di entità

Per poter misurare devo quindi avere una comprensione empirica. Devo assegnare dei valori a degli attributi di un oggetto e le relazioni che ho fra le misure, ovvero i valori che assegno agli attributi degli oggetti devono corrispondere a relazioni empiriche

20.9 Scala

Dato un sistema di relazioni empiriche Σ , un sistema di relazioni numeriche τ e una misura M , si definisce scala la terna: (Σ, τ, M) . Una scala di misura collega tra loro relazioni empiriche e relazioni numeriche. Ci deve essere la misura, ovvero il modo in cui misuro, e deve esserci anche la corrispondenza tra relazione empirica relazione numerica. Una relazione binaria è un sottoinsieme del prodotto cartesiano $A \times B$. Una relazione unaria è un sottoinsieme di un insieme. Tutti gli elementi che soddisfano la relazione sono un sottoinsieme dell'insieme di partenza. Relazione ternaria è data da $A \times B \times C$

20.10 Trasformazioni di scala ammissibili

Una trasformazione di scala ammissibile è una funzione che trasforma una qualsiasi scala in un'altra. Con la trasformazione, deve continuare a valere che il sistema di relazioni numeriche rispetti il sistema di relazioni empiriche. Il "tipo" di una scala è determinato dall'insieme di trasformazioni ammissibili.

20.11 Tipi di scale

Esistono diversi tipi di scale:

- **nominale ($M' = F(M)$):** è una scala in cui ad ogni elemento assegniamo una label. Ogni attributo è classificabile mediante un insieme di possibili valori. Essa ci consente di fare delle classificazioni. Non c'è alcuna relazione tra gli elementi della scala, per cui l'unica trasformazione F che posso fare è una trasformazione biettiva, ovvero cambiare la label.
- **ordinale ($M' = F(M)$):** dati gli elementi della scala, posso dire quale sia maggiore dell'altro, avendo un ordinamento tra i valori ma non avendo una distanza tra di essi. Ogni attributo è caratterizzato da un ordinamento lineare dei valori. Il tipo di trasformazione F che posso fare è una monotona, ovvero una trasformazione che mi mantiene l'ordinamento tra i valori
- **intervallo ($M' = aM+b$):** si ha una scala ordinale in cui sappiamo però quale sia la distanza tra i valori. Se esiste anche un concetto di "distanza relativa tra i valori" abbiamo una scala intervallo. Si parla di distanza relativa in quanto la trasformazione mi può cambiare la distanza ma la distanza relativa è immutabile. Il tipo di trasformazione che posso fare è una traslazione, in cui non si lascia inalterata la distanza ma rimane invariato il rapporto tra le distanze delle misure. Lo stesso rapporto che esisteva tra le distanze dei valori prima, rimane invariato dopo. In una scala intervallo non esiste il concetto di rapporto tra misure, poiché posso avere uno spazio di scala in cui si hanno valori negativi e positivi
- **rateo ($M' = aM$):** Se esiste anche un elemento "zero" siamo in presenza di attributi. Nella scala rateo le misure sono tutte positive. Non posso avere uno spazio di scala che mi fa diventare delle misure negative. Nella scala rateo esiste uno "zero" ma si parla di "zero" assoluto. In tal caso si può parlare di rapporti tra misure. Tutte le

trasformazioni tra scala fanno in modo che il rapporto tra le misure sia invariabile.
Possiamo fare una trasformazione che lascia inalterata la distanza tra gli elementi

- **assoluta ($M' = M$): non possiamo fare alcuna trasformazione.** Essa è una scala **immutabile** in quanto è una scala in cui conto gli elementi e basta. Il conteggio è quello e non posso cambiarlo

Differenza sostanziale tra rateo e intervallo è che nell'intervallo è immutabile (resta invariato) **il rapporto tra le distanze delle misure.** Nel rateo è immutabile (resta invariato) **il rapporto tra le misure**

Trasformazioni possibili	Tipo scala	Esempi
$M' = F(M)$ F trasf. biettiva	Nominale	Label/classificazioni
$M' = F(M)$ F trasf. monotona	Ordinale	Preferenza, test di intelligenza
$M' = aM + b$ $a > 0$	Intervallo	Temperature
$M' = aM$ $a > 0$	Rateo	Durate, lunghezze
$M' = M$	Assoluta	Entità numerabili

Figura 20.1: Tipi di scale



20.12 Sensatezza di affermazione

In base alla scala possiamo dire se un'affermazione è sensata o meno. Una affermazione relativa ad una misura è sensata se e solo se la sua verità resta invariata a fronte di una trasformazione ammissibile di scala.

20.13 Tipi di scale ed operazioni ammesse

Il tipo di scala di una misura determina l'insieme di operazioni ammesse. Nelle scale nominali e ordinali possono essere effettuati solo test statistici non parametrici: percentile, mediana. Non è possibile calcolare la media in quanto essa non ha senso. Mentre il valore della mediana cambia, quello che cambia è l'elemento, l'osservazione. Nelle scale intervallo e rateo si ammettono test statistici parametrici: media, deviazione standard, media geometrica.

20.14 Misure indirette e trasformazioni di scala ammissibili

Una misura indiretta di un attributo è una misura ottenuta misurando altri attributi: $M=f(M_1, \dots, M_n)$. Una trasformazione di scala ammissibile per una misura indiretta M è una trasformazione T tale che: $T(M)=f(T(M_1), \dots, T(M_n))$. La trasformazione deve essere quindi uguale alla funzione applicata alla trasformazione in scala di tutti gli attributi. Se non vale tale proprietà la trasformazione di scala non è ammissibile.

20.15 Misura del software

Dalla teoria della rappresentazione della misura abbiamo quindi imparato che prima di misurare bisogna decidere quale entità bisogna prendere in considerazione, quali attributi di questa entità misurare. Per applicare tutto ciò è necessario un modello concettuale (framework) per la misura del software. Nel software quello che possiamo misurare sono entità (processi, prodotti, ovvero output dei processi, e risorse, input dei processi) o attributi, divisi in interni ed esterni.



20.16 Attributi interni ed esterni

Essi sono:

- **attributi interni:** Sono gli attributi di una entità che possono essere misurati a partire dalla sola entità, ovvero mediante misurazioni dirette
- **attributi esterni:** Sono gli attributi di una entità che devono essere misurati in termini di come l'entità è in relazione con l'ambiente esterno. Sono i più importanti, in quanto i requisiti funzionali sono espressi mediante attributi esterni, ma sono anche i più difficili da misurare, in quanto non possono essere misurati direttamente. Avrò quindi bisogno di misure indirette

Riuscendo a trovare una relazione esatta tra attributi interni ed esterni, posso misurare gli attributi esterni costruendo un modello che mette in relazione attributi esterni ed attributi interni e, sulla base degli attributi interni, posso predire il valore degli attributi esterni. Esso è un modello predittivo in quanto misuro gli attributi interni durante lo sviluppo, mentre gli esterni potrò misurarli in altro modo solo alla fine. Potrò quindi costruire un modello predittivo che sulla base degli attributi interni mi predice un attributo esterno. È necessario però costruire un modello predittivo.



	ENTITA'	ATTRIBUTI INTERNI	ATTRIBUTI ESTERNI
Prodotti	Specifico	Dimensionalità, ruolo, modularizzazione funzionale, correttezza sintattica	Comprendibilità, manutenibilità
	Progetto	Dimensione, ruolo, accoppiamento, coesione, modularizzazione funzionale	Qualità, comprensibilità, manutenibilità
	Codice	Dimensione, ruolo, accoppiamento, modularizzazione funzionale, complessità algoritmica	Affidabilità, manutenibilità, usabilità
	Dati test	Dimensione, copertura	Qualità, affidabilità
Processi	Specifico	Durata, sforzo, n. cambiamenti nei requisiti	Qualità, costo, stabilità
	Progettazione	Durata, sforzo, numero di errori individuati	Costo, stabilità
	Test	Durata, sforzo, numero di errori trovati	Costo, controllabilità
Risorse	Personale	Eta, costo	Produttività, esperienza
	Team	Dimensione, livello di comunicazione, strutturazione	Produttività
	Software	Costo, dimensione	Affidabilità
	Hardware	Costo, prestazioni	Affidabilità

Figura 20.2: Attributi interni ed esterni

Esempio: se una scala è ordinale o nominale posso applicare solo classificazione. Per valutare la manutenibilità del sistema posso misurare lo sforzo richiesto nelle operazioni di manutenzione. Posso però mettere in relazione caratteristiche interne rispetto allo sforzo. Posso fare regressione e calcolare lo sforzo esatto. Il costo è un attributo esterno mentre lo sforzo non lo è in quanto il costo è calcolabile dallo sforzo e dal salario delle persone

20.17 Misure: valutazione e stima

Si misura per valutare le caratteristiche di un oggetto usando la teoria della rappresentazione della misura, ma anche per stimare le caratteristiche di un oggetto non ancora esistente. Posso, usando altre misure, stimare il costo di un oggetto non ancora esistente, costruendo un modello predittivo. In entrambi i casi possiamo utilizzare un “modello” delle caratteristiche dell’oggetto che evidenzia relazioni esistenti tra diversi attributi dell’oggetto.

20.18 Sistemi predittivi

Un sistema predittivo è composto da un modello e da un insieme di procedure per predire i parametri incogniti del modello.

Esempio: COCOMO è un sistema predittivo caratterizzato da un modello ($E = a S^b$) e da un insieme di procedure per predire i parametri a, b ed S

20.19 Sistemi predittivi e misure

è necessario distinguere tra misure, usate ai fini di valutazione, e sistemi predittivi. Nel primo caso dovremo ricercare certe caratteristiche rispetto la teoria della rappresentazione, nel secondo caso altre come la bontà della stima effettuata. Quando stiamo misurando, a noi interessa che la misura sia valida dal punto di vista della teoria della rappresentazione

della misura, per cui si abbia la **relazione tra relazione empirica e relazione numerica**. Quando sto predicendo mi interessa che il modello funzioni bene

Esempio: i function point non sono niente rispetto la teoria della rappresentazione, ma sono buoni predittori del costo di sviluppo

20.20 Comprendere la realtà e poi misurarla

Spesso, non comprendendo cosa stiamo misurando, si commettono errori. Spesso errori di misura derivano dal non aver individuato esattamente l'oggetto della misura (prodotto, processo, risorsa) e le caratteristiche dell'attributo che si misura (interno o esterno). Posso valutare le caratteristiche esterne di un prodotto indirettamente, attraverso misure di attributi di processo. è quindi molto importante capire se un attributo sia di processo e di prodotto quindi poichè, nel caso sia un attributo di processo è necessario prendere in considerazione altri fattori.

Esempio: si consideri l'attributo “numero di malfunzionamenti individuati durante la fase di test”. Tale attributo viene talvolta considerato un attributo di processo, tal altra un attributo di prodotto (numero di errori del programma). Si osservi che il primo è un attributo esterno (dipende dalla durata della fase di test), il secondo un attributo interno. Misurare le performances attraverso l'esecuzione è un attributo di processo, poichè dipende da come ho definito i casi di test.

20.21 Progettazione degli esperimenti ed analisi dei risultati

Un esperimento di misura non può essere improvvisato:

- Occorre individuare ed isolare ogni aspetto che possa influire sui risultati
 - Esistono diverse linee guida che ci consentono di progettare al meglio gli esperimenti
- è necessario **analizzare i risultati usando metodi statistici che possono essere utilizzati in base al tipo di scala della misura**

20.22 Esperimenti di misura

Gli esperimenti di misura vengono fatti quando si vuole valutare l'impatto su qualcosa. Un esperimento di misura si realizza per valutare gli effetti di certe procedure su certi soggetti. Distinguiamo quindi tra:

- **variabili indipendenti:** la misura delle azioni compiute sull'oggetto
- **variabili dipendenti:** la misura degli attributi del soggetto che si ritiene possano essere influenzate dalle azioni compiute e quindi dalle variabili indipendenti

Quello che vogliamo vedere è se la variabile indipendente ha un effetto sulla variabile dipendente.

Esempio: Si supponga di voler valutare se: la produttività dei lavoratori è influenzata dalla intensità delle luci. In questo caso metto a sviluppare persone con luce più alta ed altre con luce più bassa. Devo però preoccuparmi degli sviluppatori che hanno un effetto sulla produttività, come esperienza degli sviluppatori, conoscenza del linguaggio... Ci sono diverse strategie, come quella casuale

20.23 Strategie di sperimentazione

Durante un esperimento si sottopongono soggetti diversi a "trattamenti" per cercare di mettere in evidenza un rapporto causale tra variabili dipendenti e indipendenti. Un aspetto fondamentale della progettazione dell'esperimento consiste nello scegliere quali soggetti sottoporre a quali trattamenti al fine di isolare possibili elementi di influenza esterni. Esistono diverse strategie che suggeriscono come effettuare tale selezione. Esse sono:

- **Assegnamento casuale:** I soggetti sono assegnati ai diversi gruppi (sottoposti a trattamenti diversi) in maniera casuale. È attuabile ma bisogna poi verificare che non ci siano stati altri fattori che hanno impattato il risultato. Facendo in maniera casuale devo andare a valutare a posteriori l'esperimento
- **Assegnamento a gruppi:** I soggetti simili (che si prevede rispondano in maniera simile) sono prima raggruppati, quindi i diversi gruppi sono assegnati in maniera casuale ai diversi esperimenti

Durante la fase di analisi statistica, le differenze tra i vari gruppi vengono tenute in considerazione per isolare l'influenza delle sole variabili indipendenti.

20.24 Strategie di accoppiamento

Una volta divisi in gruppi, le strategie di accoppiamento sono:

- **Pre-accoppiamento:** assegno persone di uno stesso gruppo ai vari esperimenti, ai vari trattamenti

- **Auto-accoppiamento:** Ogni soggetto viene sottoposto a tutti i trattamenti (è come se appartenesse a tutti i gruppi). Questo dovrebbe fare in modo che un fattore esterno non incida. Devo sempre verificare però che non ci siano fattori esterni però che vanno ad intervenire. Bisogna fare attenzione che ogni soggetto non riceva i diversi trattamenti nello stesso ordine. Facendo i trattamenti in un determinato ordine, è possibile che ciò influisca. Ciò a causa non dei trattamenti, ma del learning effect. Se io faccio un qualcosa con un determinato trattamento, e faccio la stessa cosa una seconda volta con quel trattamento, sarò più bravo, poiché quel task l'ho già compiuto, a prescindere dal trattamento. Devo far fare task diversi tra i trattamenti quindi ma con la stessa complessità. Deve però cambiare anche l'ordine dei trattamenti, per evitare che ciò possa influire nel risultato. Dati due trattamenti devo quindi avere che lo stesso task è sviluppato con entrambi i trattamenti. Nel caso in cui si abbiano due task invece, i task vanno sviluppato con tutti e due i trattamenti e l'ordine dei trattamenti deve essere opposto nei due casi.

Esempio: Si supponga di voler valutare l'impatto del linguaggio di programmazione adottato sul tempo di sviluppo. Pre-accoppiamento: Si prendono progetti simili per dimensione, complessità, esperienza del team e li si sviluppa ognuno con un linguaggio diverso. Autoaccoppiamento: Si sviluppa ogni applicazione più volte, ogni volta con un linguaggio diverso e si confrontano i risultati. Come detto, occorre evitare di adottare i diversi linguaggi sempre nella stessa sequenza (prima C poi C++ ecc.) per eliminare l'influenza dell'ordine tra i linguaggi.

20.25 Analisi dei risultati

Durante la fase di analisi devo andare a vedere se c'è un fattore di causalità tra fattore principale e le variabili dipendenti e se ci sono rapporti causalità tra fattori di confusione e variabili indipendenti. I fattori di confusione a volte potrebbero non avere un impatto diretto ma interagiscono con il fattore principale. Si possono usare test statistici per capire l'interazione tra fattori. Non si va a valutare come il singolo fattore impatta sulla variabile dipendente ma piuttosto si va a vedere l'interazione tra vari fattori. Il fatto che un fattore abbia un effetto su una variabile potrebbe essere semplice correlazione. Ci può essere una correlazione ma non ci può essere causalità. Se c'è causalità, sicuramente c'è correlazione. La correlazione quindi può essere un indicatore di causalità. Per applicare i test parametrici

(media, varianza) è necessario avere una distribuzione normale. Esistono dei test per testare la normalità della distribuzione.

20.26 Validazione delle misure del software

Per il software sono state proposte numerose misure per diversi attributi. Spesso esistono molte misure diverse per lo stesso attributo. Occorre una metodologia che ci permetta di valutare la bontà di tali misure. Una misura è valida se caratterizza accuratamente l'attributo. Un sistema predittivo è valido se permette una stima accurata. Ho diversi modi per misurare l'accuratezza di una stima.

20.26.1 Validazione di un sistema predittivo

La validazione di un sistema predittivo è il processo attraverso il quale si stabilisce l'accuratezza della stima fornita dal sistema. Si va ad effettuare un confronto dei risultati del sistema predittivo con i valori effettivi a posteriori. Talvolta la validità di sistemi predittivi complessi non è migliore di quella di sistemi più semplici.

20.26.2 Validazione di una misura

La validazione di una misura è il processo attraverso il quale si stabilisce se la misura fornisca una appropriata rappresentazione numerica dell'attributo. La validazione della misura si basa sempre sulla teoria della rappresentazione. Bisogna quindi trovare corrispondenza con le relazioni empiriche

Esempio: una valida misura dell'attributo "lunghezza" di un programma non deve contraddirsi alcuna nozione empirica riguardante la nozione di lunghezza. Ad esempio se $L(m)$ è la lunghezza del programma m deve valere la formula: $L(m_1; m_2) = L(m_1) + L(m_2)$

20.26.3 Validazione di una misura: errori

Nel campo del software si afferma spesso che una misura per essere valida deve essere un accurato predittore di un qualche attributo di interesse generale. Per dimostrare la validità di una misura si cerca allora una correlazione con un qualche attributo "utile", ma questo non rispetta la teoria della rappresentazione della misura. Esempio di ciò è il caso dei function points, LOC e complessità. In questi casi, si tira fuori un qualcosa che non ha senso dal punto di vista empirico, dal punto di vista di teoria della rappresentazione della

misura. Tali attributi "utili" sono estratti senza avere in mente cosa mi serva, perchè lo si misura, quale sia la caratterizzazione empirica di tale attributo ritenuto "utile" e quale sia una ragionevole misura di tale attributo.

20.27 Caso del numero ciclomatico di McCabe

Il numero ciclomatico di McCabe misura il numero di cammini linearmente indipendenti di un programma. Esso viene ritenuto senza alcun fondamento empirico una valida misura della complessità di un programma. Essa non ha alcun fondamento empirico dal punto di vista della complessità. Programmi con cicli innestati e else if in cascata potrebbero avere la stessa complessità secondo il numero ciclomatico di McCabe, eppure il programma con ciclo innestato risulterebbe più complesso. In altri casi usando il numero ciclomatico di McCabe, un programma di base più semplice potrebbe avere un valore maggiore rispetto ad un programma più complesso. Non si può quindi considerare il numero ciclomatico di McCabe come una misura valida di complessità.



20.28 Raccolta ed analisi dei dati

Il processo di raccolta dei dati è fondamentale per garantire risultati accurati. Prima di iniziare un'attività di raccolta di dati bisogna avere chiaro in mente cosa vogliamo misurare e come (soprattutto nel caso di misure indirette). Occorre decidere come raccogliere i dati, quando raccoglierli, come conservare i dati e come estrarli quando utile e come analizzare i dati.

20.28.1 Come raccogliere i dati

I dati possono essere raccolti in maniera automatica soprattutto se si ha bisogno di dati sul prodotto software, potendoli tirare fuori dati dal codice. Posso invece tirare informazioni dalle repository che utilizzo per le metriche di processo. In ogni caso preliminare occorre decidere:

- quali entità e quali attributi misurare
- come effettuare le misure indirette (modelli)
- progettare le form per la raccolta manuale o le procedure per la raccolta automatica
- progettare le procedure per l'analisi dei dati e la presentazione dei risultati

20.28.2 Quando raccogliere i dati

La raccolta dei dati dipende fortemente dall'entità e dall'attributo che occorre misurare. In generale, i dati vengono raccolti durante l'intero ciclo di vita. Per alcuni dati, la raccolta prosegue anche dopo il rilascio del prodotto.

20.28.3 Come conservare i dati

I dati vanno conservati in un database. Solo un database fornisce gli strumenti per effettuare estrazioni efficienti. Inoltre l'uso di database riduce i problemi di accesso/modifica concorrente dei dati. Non è fondamentale il modo in cui implemento il database. Non è infatti fondamentale il fatto che il database sia relazionale. L'importante è che ci sia uno strumento di archiviazione di dati che consenta di estrarli facilmente. Da questo dipende la successiva possibilità di effettuare analisi e classificazioni accurate.

20.28.4 Come analizzare i dati

Per l'analisi dei dati ci serviamo dei metodi statistici. Non tutte le analisi statistiche sono applicabili a tutte le misure. Ciò dipende dal tipo di scala adottata. Molte analisi statistiche sono applicabili solo a distribuzioni normali mentre molte misure nel campo del software non hanno distribuzione normale. Si può quindi ricorrere ad analisi non parametriche o a statistiche "robuste". Posso, inoltre, variare la scala per avvicinarmi a una distribuzione normale.

20.29 Statistiche

Si parla di distribuzioni con un numero discreto di elementi. Alcuni esempi potrebbero essere media e varianza.

20.29.1 Box Plot

Se le distribuzioni sono non normale o sono su scala non rateo, possiamo usare il box plot. All'interno del box plot sono individuati:

- **mediana (m):** valore del dataset che lo divide in due insiemi equipotenti (a destra e sinistra c'è lo stesso numero di punti)

- **upper fourth (u)** e **lower fourth (l)**: mediane dei valori maggiori di m e inferiori di m . Rappresentano i valori del primo quartile e del terzo quartile. U , m ed l dividono in 4 parti uguali il dataset



Figura 20.3: Box Plot

Altre misure sono:

- La lunghezza del box plot è data da $u-l$.
- **uppertail teorico:** $u+1.5d$
- **lowertail teorico:** $l-1.5d$

Uppertail e lowertail non sono la fine della distribuzione ma rappresentano lowertail ed uppertail teorico, ovvero è ragionevole considerare i valori che si trovano nell'ambito di una distanza di $1.5d$. Tutto ciò esterno viene considerato un outlier. I valori della distribuzione potrebbero non essere presenti nella distribuzione, per cui si approssima all'elemento del dataset più vicino.

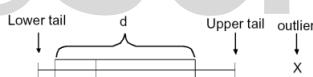


Figura 20.4: Box Plot

20.30 Relazione tra due attributi

20.30.1 Regressione lineare

Per analizzare la relazione tra due attributi possiamo usare uno scatterplot. Lo scatterplot ci fa capire se esiste una relazione tale che esiste una curva che rappresenta i valori tra i due attributi. A volte vengono curve precise per cui si possono effettuare osservazioni migliori. Per valutare l'esistenza di una correlazione possiamo utilizzare la regressione lineare, vvero andiamo a calcolare la retta che meglio approssima l'insieme di valori. Teoricamente bisognerebbe calcolare la retta, ma potrebbero uscire funzioni non lineari.

Supponendo di fare una trasformazione sui dati, ovvero elevo i dati di x al quadrato. Ottengo così x' . Posso quindi trasformare le variabili indipendenti. Quando vado a testare

devo considerare le trasformazioni che ho fatto. Con trasformazioni di scale diverse ottengo un'equazione non lineare, applicando però regressione lineare. Quando metto il modello, devo sostituire le nuove variabili con la variabile da cui ho ottenuto la trasformazione. L'equazione della retta è $y = a + bx$. Posso avere più variabili indipendenti.

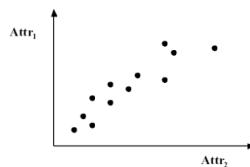


Figura 20.5: Scatterplot

20.31 Regressione logaritmica

La regressione logaritmica è usata per calcolare la curva esponenziale che meglio approssima l'insieme di valori. L'equazione della curva è $y = b * m^x$. Nella regressione lineare posso ottenere un modello non lineare applicando le trasformazioni di scala solo alle variabili indipendenti. Nella logaritmica applico la trasformazione di scala, che è la stessa, non ai dati al modello, trasformando il modello.

20.32 Parametri calcolati in Excel

Alcuni parametri sono:

- **r2: coefficiente di determinazione.** Il coefficiente di determinazione ci dice quanto bene una funzione spiega i dati. Meglio i dati fittano all'interno della curva, maggiore sarà il coefficiente di determinazione
- **sey:** l'errore standard per la stima y
- **F:** esso rappresenta la statistica F, il valore osservabile F. Essa è usata per determinare se la relazione osservata tra le variabili dipendenti e indipendenti è casuale
- **df:** grado di libertà usato per trovare valori critici di F nella tabella statistica. Il confronto tra il valore trovato nella tabella e la statistica F prodotta dalla funzione REGR LIN determina il livello di confidenza del modello.
- **ssreg:** somma di regressione dei quadrati

- ssresid: somma residua dei quadrati

Il confronto tra df e statistica ci fa capire il livello di fiducia del modello. La statistica ci dice che, con determinati gradi di libertà, possiamo essere certi ad un x% che la probabilità non sia casuale.

20.33 Misure in ambito industriale

Al fine di stabilire un **programma di misura di successo** in ambito industriale occorre tenere presenti diversi aspetti organizzativi:

- **motivazioni ed obiettivi del programma:** ogni programma di misura deve avere dei chiari e precisi obiettivi. Tali obiettivi dovrebbero essere espressi in termini di processi, prodotti e risorse ben precisi. A partire dagli obiettivi dovrebbero essere definite anche le azioni da intraprendere a valle dell'analisi dei dati raccolti per raggiungere gli obiettivi
- **responsabilità organizzative e composizione del team di misura:** per avere successo occorre avere chiaro chi abbia la responsabilità del progetto
- **attività di raccolta dei dati**
- **training e motivazione del personale:** Nessun programma di misura ha speranza di successo se il personale coinvolto non è opportunamente motivato

CAPITOLO 21

Software quality management

21.1 Introduzione: software quality management

Si effettua quality assurance per poter produrre un prodotto di qualità. Un prodotto è di qualità se rispettano i requisiti. Per ottenere un prodotto di qualità è necessario, assicurandoci quindi di avere una buona analisi dei requisiti, buona progettazione... è necessario agire sulla qualità del processo. È quindi importante definire degli standard, delle procedure di qualità ed assicurarsi che siano seguite. Quando si parla di qualità si parla di cultura della qualità all'interno dell'azienda. Avere un processo di elevata qualità, non vuol dire che il prodotto abbia un'elevata qualità. Se un'azienda è certificata ISO 9000, non significa che l'azienda faccia un prodotto buono, ma significa che l'azienda fa quel prodotto e lo fa sempre allo stesso modo. Quando si chiede una certificazione di qualità ad un'azienda, sui processi, è per avere garanzia che essa non faccia sorprese, che mi consegni ciò che mi ha promesso.

21.2 Qualità

La qualità significa che un prodotto rispetta le specifiche, che per il software possono essere un po' più problematiche. Si potrebbe avere un disallineamento tra requisiti di qualità del cliente e quello dello sviluppatore. Inoltre alcuni requisiti di qualità sono complessi da specificare in maniera non ambigua. Il problema è quindi specificare i

requisiti di qualità in modo che possano essere implementati e soprattutto verificati. Molto spesso inoltre le specifiche sono incomplete ed è necessario mettere in piedi procedure per migliorare la qualità anche a dispetto delle specifiche non precise. La qualità del prodotto non riguarda solo la riduzione di difetti ma anche altri aspetti come: **usabilità...**

21.3 Attività di quality management

Esse sono:

- **quality assurance:** indica ciò che fa un'azienda per mettere in piedi standard e procedure di qualità. L'obiettivo di un'azienda o del team di quality assurance dell'azienda, è quello di definire un manuale di qualità. Esso contiene tutti gli standard e le procedure seguire dall'azienda
- **quality planning:** un'azienda che sviluppa software è un'azienda che sviluppa progetti. Nell'ambito di tali progetti è necessario definire la qualità. Pianificare la qualità vuol dire andare a definire un piano di qualità. Esso contiene le attività che devono essere svolte, la tempistica, le risorse, ma contiene anche la parte del manuale di qualità che può essere applicata all'interno del progetto. Esso contiene le standard e le procedure che dobbiamo selezionare dal manuale e che vanno applicate all'interno del progetto. Possono esserci casi in cui delle procedure vadano customizzate per il particolare progetto, o sia necessario aggiungere qualcosa. Esso avviene all'interno dei progetti, come il quality control
- **quality control:** è necessario controllare la qualità. Serve ad assicurarsi che le procedure e gli standard siano seguiti dal team di sviluppo software. Solitamente quality manager deve essere separato dal project manager. project manager infatti predilige la velocità e non la qualità. Molto spesso, in corrispondenza di ogni derivabile ci sono delle review che vengono fatte. Insieme al documento viene quindi consegnato un report contenente il risultato della review.

21.4 ISO 9000

Insieme di standard per la gestione della qualità. Può essere applicato a diversi tipi di organizzazione, in particolare ISO 9001 è applicabile ad aziende che progettano, sviluppano

e mantengono prodotti. ISO 9000-3 riguarda invece nello specifico aziende che sviluppano software. ISO 9000 mi indica le attività che devono essere svolte, ma non mi dice come.

21.4.1 Certificazione ISO 9000

è necessario avere un manuale di qualità che è coerente con l'ISO 9000 o con lo standard ISO 9000 con cui la mia azienda si rifà. è poi necessario verificare che le cose vengano fatte in un certo modo. Devo in qualche modo ispezionare i progetti, vedere se ci sono le procedure e se le procedure sono seguite. Devono esserci anche le procedure per aspetti critici, che in qualche modo sono richieste dal sistema di qualità.

21.4.2 ISO 9000 e quality management

ISO 9000 definisce quindi dei modelli di qualità generici che vengono istanziati con un manuale di qualità aziendale che è usato per sviluppare piani di qualità dei vari progetto e documenta i processi di qualità dell'organizzazione che poi sono istanziati nell'ambito della gestione di qualità del singolo progetto. Posso usare ISO/IEC 12207 per definire come sono i processi che vado ad utilizzare. Non è soltanto il codice, ma anche il documento che deve essere standardizzato. ISO 9000 definisce caratteristiche che tutti i componenti dovrebbero seguire. Nell'ambito del coding è standardizzato anche lo stile di programmazione che deve essere usato. Gli standard di processo definiscono come il processo software dovrebbe essere eseguito.

21.5 Importanza degli standard

Gli standard vanno ad incapsulare le best practices, in modo tale da evitare che vengano ripetuti errori fatti in passato. Essi sono inoltre dei framework per il processo di quality assurance poiché consente di controllare che i processi seguano gli standard e forniscono continuità. Se arrivano persone nuove, esse possono imparare dallo standard come le cose vadano fatte. Se non abbiamo documentazione di questo tipo ed arrivano persone nuove, deve esserci qualcuno che spieghi come le cose vadano fatte, il che non è ottimale.

21.6 Problemi con gli standard

Gli standard non sempre sono visti di buon occhio in quanto limitano la libertà delle persone. Essi possono essere visti come troppo burocratici da parte degli sviluppatori. Se

non ci sono tool di supporto, il lavoro manuale da fare può essere tedioso. È necessario cercare di avere dei tool soprattutto per produrre la documentazione, report... che consente di automatizzare il più possibile la produzione dei documenti nell'ambito della gestione della qualità. Per far sì che gli standard rimangano aggiornati, è necessario coinvolgere gli sviluppatori durante lo sviluppo degli standard stessi. Molto spesso l'errore che si fa è quindi non ascoltare coloro che devono seguire lo standard.

21.7 Standard di documentazione

Gli standard di documentazione sono molto importanti. Essi si dividono in:

- **standard dei processi di documentazione:** il documento viene creato. Una volta creato esso viene approvato, poi viene pulito e prodotto, in cui il documento viene stampato e distribuito
- **standard dei documenti:** Esso racchiude standard come i documenti devono essere identificati univocamente, standard sulla struttura dei documenti (layout, font, stili, tavole), standard di presentazione dei documenti e standard sull'aggiornamento dei documenti ovvero come vanno cambiati i documenti rispetto le versioni precedenti
- **standard di scambi di documenti:** come i documenti sono salvati e scambiati tra differenti sistemi di documentazione

21.8 Qualità del processo e qualità del prodotto

La qualità del processo può avere un impatto sulla qualità del prodotto. Se io però ho definito dei processi, ovvero li ho standardizzati, la qualità del prodotto si assesterà. Se voglio migliorare il prodotto devo agire sul processo.

21.9 Process based quality

Nel caso non del software è bastanza evidente dove posso agire per ottenere un certo miglioramento. Nel caso del software non è proprio così. Si potrebbero usare tool differenti, ma non è detto che ciò migliori il prodotto o si ottenga quanto desiderato. Essendo molto manuale la creazione del software, ci sono altri fattori che possono influenzare. Fondamentalmente dovrei definire un processo, sviluppare il prodotto, valutare la qualità del prodotto

e se la qualità del prodotto è quella desiderata bene, posso standardizzare il processo ed applicarlo sempre, ottenendo prodotto di quella qualità. Se invece il risultato è non ok, devo agire sul processo, devo cambiarlo o migliorarlo. Trovato cosa migliorare, devo lanciare dei progetti in cui vado a sperimentare e devo poi poter andare a confrontare i risultati che ottengo rispetto a progetti in cui non ho fatto tale sperimentazione, potendo vedere se ho ottenuto un miglioramento di quella particolare caratteristica di qualità. Se però non so quale variabile indipendente può avere un effetto sulla variabile dipendente che mi rappresenta da un punto di vista di metriche l'aspetto di qualità che voglio andare a migliorare, diventa soggettivo il miglioramento. Dovrò quindi andare ad agire su un processo e dovrò capire quale sarà la variabile indipendente da modificare. Il problema è capire dove intervenire, capire cosa migliorare. Bisogna avere un obiettivo specifico per migliorare il processo e sulla base di tale obiettivo devo capire quali sono gli attributi che devo andare a misurare perché sono indicatori di quell'aspetto di qualità.

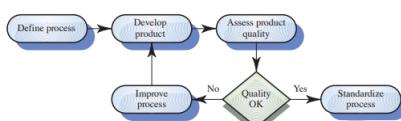


Figura 21.1: Process based quality

Esempio: il questionario che io progetto è il modo per raccogliere metriche. Facendo usare alle persone l'applicazione mi accorgo di quanto sia usabile il programma.

21.10 Practical process quality

Si definiscono gli standard, facciamo in modo che vengano seguiti e raccolgo poi dei dati. Quando raccolgo i dati, li analizzo per andare ad individuare eventuali problemi, eventuali aspetti che posso andare a migliorare. Devo sapere, in accordo ai dati, cosa sto analizzando. I dati che raccolgo devono essere lo specchio di quello che è il prodotto o il processo software. Devo quindi riuscire ad interpretare aspetti legati al prodotto o al processo software. Posso quindi individuare problemi sulla qualità del prodotto. Individuati tali aspetti posso agire sul processo, andando a modificarlo per ottenere un miglioramento di certi aspetti del prodotto.

21.11 Quality planning

Il piano di qualità, definisce le qualità del prodotto e definisce inoltre il profilo di qualità del prodotto. Si definisce il protocollo utilizzato, si definiscono inoltre i livelli di qualità. Posso definire dei valori, porli su una scala, dovendo però indicare come è definito ogni valore della scala. Costruendo un modello di qualità devo dire come arrivo a definire i valori delle caratteristiche esterne che dipendono dai valori delle caratteristiche interne. Devo anche definire il processo di assessment della qualità e devo definire gli standard che devono essere applicati. Devo quindi definire il profilo di qualità (dire come misuro, misure dirette, misure indirette), definire come faccio il controllo di qualità e poi devo definire quali sono gli standard, di prodotto e di processo, che vado ad utilizzare. Questi sono gli aspetti fondamentali del quality planning.

21.11.1 Quality plan structure

Essa è composta da: product introduction, product plans, process description, quality goals e risks and risk management. I quality plans dovrebbero essere corti, documenti succinti. Gli attributi di qualità possono essere molti, come safety, security, resilience, robustness, understandability, testability... ed ognuno degli standard ha i propri attributi

21.12 Quality control

Riguarda il controllare lo sviluppo software assicurandosi che gli standard di qualità siano seguiti. Ci sono due approcci che possono essere usati:

- quality review
- posso raccogliere automaticamente misure e controllare automaticamente

21.13 Quality review

C'è un moderatore, un gruppo esamina parti del processo e la sua documentazione alla ricerca di errori. Si organizza il processo, si hanno dei feedback e vengono fatte delle modifiche. La review può essere fatta sia sul prodotto che sul processo. Posso inoltre avere review che mi vanno a vedere l'avanzamento del lavoro, se siamo in linea con tempi e costi. Posso avere una review di qualità sia sul prodotto che sul processo.

Si fa poi un'analisi tecnica per trovare fault o mismatch tra la specifica del progetto ed il codice. La **quality review** viene fatta per approvare qualche documento che viene poi **baselined**, potendo passare all'attività successiva. Una **baseline** è un documento che è stato **formalmente approvato dal management**. Essa è stata **approvata a seguito di una review**.

Esempio: posso vedere quanto abbiamo speso ma se abbiamo anche prodotto abbastanza, essendo in linea con i tempi, costi e sviluppo. Vado a vedere rispetto ai tempi e ai costi quanto sto avanti e quanto sto indietro

21.13.1 Review functions

Review functions possono essere funzioni relative alla qualità oppure quando si fa la review per il progresso del progetto, esse sono funzionali al project managers. Posso andare ad arricchire la conoscenza aziendale, sapendo se ci sia bisogno di fare training per alcune persone. C'è il report che viene dato all'autore indicando le azioni che deve compiere.

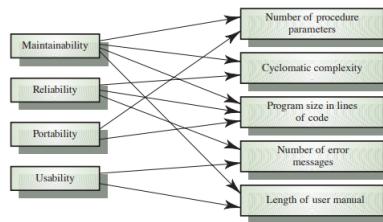
21.14 Software measurement and metrics

Il processo di misura va ad assegnare un numero o un simbolo ad un attributo di un processo di prodotto in modo tale da poter confrontare vari aspetti. Alcuni esempi di metriche sono:

- **fog index:** serve a misurare quanto complessi sono i documenti. Se esso contiene frasi molto lunghe, è complesso leggerlo
- **lines of code**

Le metriche consentono quindi di quantificare attributi di processo e di prodotto. Esse possono essere usate per predire o per controllare. Avendo definito un profilo di qualità sul mio prodotto, durante lo sviluppo posso andare a controllare se i valori siano in linea, ma posso usare le metriche anche per fare predizioni, basandoci su statistica. Durante il processo software posso quindi tirar fuori delle misure che servono poi al manager per prendere decisioni sul come cambiare il processo per allinearmi se i valori siano disallineati.

Lo standard ISO 9126 prevede maintainability, reliability, portability, usability e performances come standard di prodotto. L'ISO 9126 è fatto di caratteristiche esterne e sottocaratteristiche esterne fino ad arrivare alle caratteristiche interne e alle metriche.

**Figura 21.2:** Software measurement and metrics

21.15 Processo di misurazione

In base ai valori alle caratteristiche interne, devo definire delle scale di misura per i livelli più alti, in modo tale da mappare il valore delle metriche. Non sempre i valori sono confrontabili, per cui è necessario portare tutti i valori sulla stessa scala, portandoli tutti all'interno dello stesso intervallo. Facendo ciò ed applicando una regressione lineare, tiro fuori un modello che mi permette di misurare un aspetto del software a partire dalle caratteristiche interne (indice di Omar). È importante costruire un modello di questo tipo in quanto io posso mappare i valori su una scala della caratteristica esterna, ma per fare ciò potrebbe essere necessario avere altre informazioni che potrebbero richiedere l'esecuzione del sistema. Il processo di misurazione è parte del quality control process. I processi collezionati durante questa procedura dovrebbero essere mantenuti come una risorsa organizzazionale.

Esempio: per vedere la manutenibilità dovrei vedere la storia passata del sistema. Se non ho questi dati prendo l'indice di Omar e lo applico, ottenendo le informazioni sulla manutenibilità

21.16 Data collection

La metrica di un programma dovrebbe essere basata su un insieme di dati afferenti al prodotto ed al processo. Essi dovrebbero essere collezionati immediatamente e possibilmente, in modo automatico. Ci sono 3 tipi automatici di data collection:

- static product analysis: tiro fuori dati staticamente, dal codice o dai documenti
- dynamic product analysis: metriche che vanno a misurare attributi che fanno riferimento al processo di esecuzione del sistema, non sul prodotto. Esse possono essere usate per calcolare caratteristiche esterne del prodotto
- process data collection

Insieme al task è sempre necessario riportare lo sforzo, collezionando quindi i dati immediatamente per tenere il processo ed i costi sotto controllo.

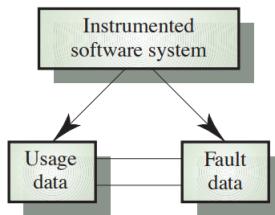


Figura 21.3: Data collection

21.17 Metriche statiche e dinamiche

Non stiamo misurando il prodotto ma bensì attributi del processo. Le metriche dinamiche sono strettamente legate al software quality attributes. Le metriche statiche hanno una relazione indiretta con gli attributi di qualità mediante dei modelli che mi riescono a mappare le misure dirette delle caratteristiche interne su misure degli attributi esterni.

Alcune metriche potrebbero essere fan-in/fan-out (fan-in alto la componente è molto accoppiata con il sistema, mentre fan-out alto la componente è molto complessa in quanto si necessita di molte componenti per eseguire le funzioni), length of code, complessità ciclomatica, lunghezza degli identificatori (più lunghi sono più descrittivi sono), profondità della condition nesting, ovvero delle strutture di controllo, fog index... Ci sono alcune metriche specifiche per la OOP come al profondità dell'inheritance tree, method fan-in/fan-out (concetto identico ma applicato ai metodi), weight methods for class e number of overreading operations.

21.18 Measurement analysis

Mediante la statistica è possibile capire il significato dei dati che si raccolgono. Ci potrebbero però essere dei controsensi come ridurre il numero di faults potrebbe aumentare il numero di chiamate ad un help desk. Questo perchè bisogna fare attenzione a come si misura. Si potrebbero avere più clienti ora rispetto a prima poichè il software è più affidabile, per cui overall sono aumentate le chiamate ma in percentuale rispetto alle persone, le chiamate sono diminuite. Oppure gli stessi utenti iniziano ad usare altri componenti trovando altri errori.

CAPITOLO 22

Software cost estimation

22.1 Introduzione: Software cost estimation

Quando si fa la **cost estimation** ciò che si va a stimare è lo sforzo richiesto per compiere un'attività o un task, per sviluppare il sistema ed anche il tempo necessario, stimato sulla base di alcune caratteristiche del sistema. Esse sono attività che vengono svolte continuamente. Si fanno delle stime iniziali, ma essendo inaccurate, durante il ciclo del progetto ne verranno fatte di altre per cui, avendo a disposizione più informazioni, risulteranno più accurate avvicinandosi di più a costi e tempi effettivi del progetto. È quindi un qualcosa che viene fatto di continuo. Così come si rivede la pianificazione si rivede la stima dei costi di un progetto.

È importante stimare i costi in quanto non è possibile partire con un progetto senza tale stima. Ciò è importante per il personale del progetto, che vuole capire se il budget ed il tempo sono realistici. È importante per gli analisti ma anche per i project manager che devono coordinare le attività di un progetto, dovendo contabilizzare i costi delle varie attività.

È importante avere stime corrette in quanto stime scorrette possono portare una serie di problemi al cliente che aspetta un software che viene consegnato in ritardo, allo staff che deve correre ai ripari... Si potrebbero spostare soldi da un progetto all'altro creando effetti a catena di software che stanno andando male.

22.2 Componenti di costo

Andando a fare una stima di costi, i componenti che devo considerare sono:

- **hardware e software:** devo fare una stima sulle risorse in termini di apparecchiature che andrò ad utilizzare
- **viaggi:** dovrei magari spostarmi dal cliente
- **addestramento:** dovrebbe essere necessario fare **training**
- **consumables:** oggetti consumabili come penne, carta...
- **effort cost:** questa è la **componente predominante**, ovvero lo sforzo del personale che si traduce in stipendi, salario.
- **overhead:** costi indiretti. Si potrebbero avere costi di servizi ovvero costi di terze parti, ma anche costi di riscaldamento, elettricità...

Tutti tranne l'overhead sono costi diretti, ovvero costi imputabili direttamente al progetto, spese che faccio per il progetto e direttamente per quel progetto. Costi indiretti sono tutti quei costi che non sono direttamente imputabili al progetto, nel senso che non riesco a quantificare direttamente quanto di quei costi spendo per il progetto. I costi indiretti vanno allocati su tutti i progetti e l'allocazione dipende dal numero di persone che ci lavorano. I costi indiretti si stimano quindi su una percentuale dei costi diretti del personale. Si fa maggiormente riferimento al personale in quanto sono le persone che consumano le altre risorse, per cui più persone si avranno più i consumi saranno alti. Il budget viene quindi fatto per stimare costi di un progetto, dovendo però considerare anche il profitto, che verrà definito in termini di margine di profitto.

22.3 Costing and pricing

Costing di sviluppo e pricing sono due oggetti differenti, per cui è difficile definire una relazione. Possono esserci ragioni differenti che vanno a influenzare il modo di fare il pricing, come:

- **market opportunity:** per aggiudicarmi la gara potrei fare un pricing minore, per cui avrei meno profitto

- **incertezza stima di costi:** faccio una stima di costi incerta, per cui per non perderci aggiungo una contingenza, metto un'extra budget all'interno dei costi. Aumentando i costi anche il prezzo sarà più alto. La contingenza la metto sui costi in modo tale da avere un budget dei costi entro il quale riesco a rientrare per sviluppare il progetto
- **termini contrattuali:** sviluppando un sistema per un cliente, esso viene sviluppato in partnership. Potrebbero esserci regolamentazioni contrattuali differenti dal solito lasciare i diritti del software al cliente insieme alle spese necessarie.
- **volatilità dei requisiti:** se i requisiti è molto probabile che cambino, si può abbassare il prezzo per vincere il contratto, andando però a mettere dei sovrapprezzati nel caso in cui i requisiti cambino
- **problemi finanziari:** se si hanno problemi finanziari si potrebbero alzare i prezzi. è quindi meglio lavorare ed andare in perdita, piuttosto che non lavorare. In tal caso infatti si andrebbe in bancarotta.

22.4 Produttività del programma

Misura del rate con cui gli sviluppatori producono software e documentazione associata. Non è una metrica quality oriented in quanto non considera la qualità. In alcuni casi potrebbe essere un problema in quanto bisogna sempre pesare la qualità e non solo la produttività. Essenzialmente si vuole misurare la funzionalità prodotta per unità di tempo.

22.5 Misure di produttività

Di solito le misure di produttività sono size related, legate a linee di codice, pagine di documentazione... ma possono anche essere function related. Essa è una stima delle funzionalità sviluppate di un prodotto software. Le metriche basate su size/tempo non hanno in conto della qualità, e molto spesso la produttività può aumentare ma a discapito della qualità. è quindi necessario fare attenzione ad entrambe.

22.6 Stima della size di un software

Quello che si può fare è stimare la size di un prodotto, stimare il numero di programmatore, stimare la produttività di un contratto. Per stimare la size di un software è necessario

definire la **minimum estimated size (a)**, la **maximum estimated size (b)** e la **most likely estimated size (m)**. Si avrà poi l'**expected estimation (E)** e la **standard deviation (alpha)**.

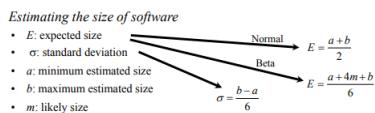


Figura 22.1: Problemi di misura

Nell'ambito della distribuzione m ha un peso maggiore rispetto minimo e massimo. è comunque necessario fare una stima massima e minima della size del software. Sarà necessario calcolare una size probabile del software, per cui **minimo e massimo** mi indicheranno la varianza rispetto al **most likely**. Per le stime massime e minime verranno presi come riferimento progetti differenti. Di solito si fa infatti una stima per analogia ovvero per riferimento, basandosi su progetti simili.

22.7 Linee di codice

Si parla di LOC poichè esse definiscono uno statement. Tale misura è stata definita quando ogni statement andava in una linea. In tali linguaggi di programmazione ha senso parlare di LOC, mentre in altri potrebbe essere necessario parlare di misure più forti o sensate. Sarebbe inoltre necessario definire quali programmi del software dovrebbero essere contanti. Il testing per esempio, potrebbe essere non misurato mediante LOC. Si potrebbe infatti derivare la quantità di codice di test sulla base della quantità di codice che sviluppo. Sulla produttività potrebbero esserci alcuni problemi da considerare. Il linguaggio in cui si va a sviluppare potrebbe portare a preferire la produttività di uno sviluppatore piuttosto che un altro solo perchè si usa un linguaggio più semplice o più compatto rispetto di un altro.

Usando quindi modelli algoritmici in cui la funzione di costo è derivata attraverso inferenza statistica e a partire dalle caratteristiche del codice, size inclusa, tutto ciò che può essere derivato da tali caratteristiche, dovrebbe essere incluso nel modello. Le variabili incluse nel modello non dovrebbero essere correlate. Mettendo variabili correlate, il modello non funzionerebbe bene. Essendo relazione tra codice di test e codice, il codice di test non è inserito. Può esserci una relazione tra linee di codice e documentazione, ovvero avendo software più grande si presume che si abbia più documentazione, avendo una correlazione lineare tra documentazione e codice. Essendoci tale correlazione sarebbe inutile

mettere anche la documentazione nel modello di stima dei costi. Si deve quindi evitare la correlazione tra variabili indipendenti del modello.

22.8 Function points

Il linguaggio ad alto livello aumenta la produttività, mentre più bassa è la tecnologia, più bassa sarà la produttività. Questo accade con i LOC. Tale problema è però risolto usando i function point, che sono più indipendenti dal linguaggio di programmazione, pur non essendo una misura della funzionalità. Usando la stessa tecnologia, la produttività dipende dallo sviluppatore e i function points possono essere non usati, ma usando tecnologie differenti, è necessario definire quali sono i fattori che influenzano la produttività.

I function points sono quindi basati su input ed outputs, user interactions, external interfaces e files usabili dal sistema. Ad ognuna di esse è affidato un peso. La count di function points è computata moltiplicando ogni count con il peso. Il conteggio grezzo è oggettivo mentre il peso è soggettivo. Sulla base delle 5 caratteristiche, dei vari livelli di peso, è necessario aggiungere 14 caratteristiche di aggiustamento per avere dei valori che si avvicinino al risultato di stima dei costi.

Ci sono delle regole che consentono di trasformare i function points in linee di codice, per cui si potrebbero calcolare i LOC. Questo è necessario perché i modelli algoritmici di stima dei costi, stimano i costi usando le LOC. Si calcolano quindi le function points, le si trasformano in LOC e si usano le LOC, in quanto calcolare le LOC sarebbe più complesso.

22.9 Object points

Function-related measure quando si usa un linguaggio 4GLs o similari. Essi non sono gli oggetti delle classi. Essa è una stima pesata di:

- numero di schermate separate
- numero di report
- numero di moduli 3GL che devono essere sviluppati per supplementare il codice 4GL

Tali object points posso anche usarli duranti le fasi iniziali del progetto, come nella fase di requirements elicitation. I function points possono essere utilizzati tra la fase di e la fase di design. Non sarà più una stima dei function points ma sarà un conteggio effettivo.

Con gli object points invece, si avrà un conteggio effettivo già alla fine della requirements elicitation.

22.10 Fattori che influenzano la produttività

Ci sono vari fattori che influenzano la produttività:

- **esperienza del dominio applicativo:** se si ha esperienza con il dominio applicativo, l'effort sarà minore
- **process quality:** il fatto che lo sviluppo è standardizzato porta ad una maggiore produttività, riducendo i costi
- **size del progetto:** la size del progetto e la size del team potrebbero influenzare la produttività. Maggiore è la grandezza del team, maggiore potrebbe essere il tempo impiegato tra la comunicazione delle persone, parlando di diseconomia di scala. Crescendo la dimensione del sistema, i costi aumentano in maniera più che proporzionale. Aumentando in maniera meno che proporzionale invece si avrebbe un'economia di scala. Con la diseconomia di scala 2 mi costa più di 1+1 ciò proprio dal fatto che si avranno team più grandi
- **supporto tecnologico:** il supporto sulla tecnologia potrebbero avere un impatto sulla produttività
- **ambiente di lavoro:** un ambiente più tranquillo è più produttivo

22.11 Tecniche di stima dei costi

Tecniche però per definire stime dei costi potrebbero essere:

- **giudizio degli esperti:** uno o più esperti sia della tecnologia che dell'application domain usano l'esperienza per predire i costi. Di solito è un processo iterativo, cercando di arrivare ad un consenso. Una tecnica di questo tipo è la **tecnica Delphi** che cerca di raggiungere il consenso di gruppo. I problemi di Delphi è che se ci sono due persone A e B, dove A è una persona con maggiore esperienza, è più probabile che si darà ragione A anche se non è detto che A abbia ragione. Per risolvere tale problema si è definita la **Wideband Delphi**, ovvero si ha una **valutazione anonima**. Si aprono le valutazioni anonime e si ragiona. **Tale processo si rifà fin quando non si**

converge verso una valutazione unica. Essa è molto economica e può essere accurata. Lo svantaggio è che è necessario avere esperti

- **leggi di parkinson:** il progetto costa quanto abbiamo. Il vantaggio è che sicuramente non si spende oltre, mentre lo svantaggio potrebbe essere che non si arriva allo sviluppo, potrei metterci più di quanto serve. Se il progetto costasse x e io metto $2x$, io spenderei $2x$ con tale tecnica. Potrebbe però sottovalutare anche il costo dei progetti e non necessariamente sopravvalutarlo.
- **pricing to win:** progetto costa quello che il cliente è propenso a spendere. Il vantaggio è che si vince il contratto, lo svantaggio è che la probabilità che il cliente riesca ad ottenere ciò che vuole è molto bassa.
- **stima dell'analogia:** essa si combina spesso con la stima degli esperti. Il costo di un progetto è basato su software precedenti che fanno operazioni simili. È accurato, ma i software precedenti potrebbero non essere disponibili
- **algoritmi cost modeling:** essi vanno a stimare lo sforzo, tempo di sviluppo, produttività e l'average staffing, FSP che è il rapporto tra sforzo e durata. Esso mi indica quante persone in media lavorano su un progetto. C'è un algoritmo che serve a calcolare e che si basa su modelli tirati fuori da regressione lineare, modelli che a partire da certe caratteristiche stimate, vanno a calcolare lo sforzo

22.12 Tecniche di stima dei costi: Problemi

Non è facile fare una stima dei costi, in quanto:

- **stime iniziali che vanno riviste.** In mancanza di dati sufficienti essere potrebbero essere sbagliate
- **non si sa quali persone debbano lavorare al progetto**
- **stime di costi auto-riempenti,** ovvero si fa la stima e si definisce il budget e si sviluppa quello che posso sviluppare con quel budget

22.13 Stima top down e bottom up

Con tutti questi approcci può essere usata la stima botto-up e top-down. Entrambi utilizzano la work breakdown datastrcutre del progetto. Nella top down si stima prima

il costo complessivo andando poi a distribuire il costo tra i vari elementi, mentre nella bottom up si fa il contrario. Si stimano i costi dei singoli task e poi i costi si aggregano sui nodi interni fino ad arrivare alla radice, ovvero l'intero sistema

22.13.1 Stima top down

Essa può essere utilizzata quando si ha una work breakdown datastructure non molto dettagliata, per cui si stimano i costi e poi si cerca di distribuirli. Di solito si tengono in considerazione tutti i costi, ma non avendo molto dettaglio, potrei stimare erroneamente dei costi.

22.13.2 Stima bottom up

Avendo un'architettura completa e la work breakdown data structure, ovvero i singoli task ed i singoli elementi che devono essere sviluppati del mio sistema, posso in qualche modo stimare i costi. Devo stare però attento a non sottostimare i costi di integrazione. Avendo una work breakdown data structure program oriented, si riuscirebbe a stimare i costi degli elementi ma non delle attività trasversali.

22.14 Work Breakdown Structure

Quella che conviene utilizzare è una work breakdown data structure che prende in considerazione sia l'activity che il product, ovvero presenta gerarchia del prodotto e gerarchia dell'attività. Avendo entrambe le work breakdown data structure, posso creare una che interconnette le due gerarchie.

Uso la work breakdown data structure perchè con essa posso definire i costi per le varie attività, ma anche come base per raccogliere e produrre report dei costi del software. Alloco risorse e budget per ogni elemento della work breakdown data structure, dopodiché devo andare a monitorare quanto ho speso per un'attività e se è completata o meno. Ciò può essere fatto per ogni attività. Mantenendo tali informazioni in un sistema di gestione delle informazioni, ho la possibilità di controllare tempi e costi rispetto al budget disponibile.

22.15 Metodi di stima

Ogni metodo ha punti di forza e di debolezza. La stima dovrebbe essere basata su metodi differenti. Se metodi diversi mi danno stime diverse vuol dire che c'è qualcosa che

non va. A volte pricing to win è l'unico metodo, ovvero nel caso di fallimento dell'azienda se non si riesce ad avere un contatto. La stima dei costi è principalmente experience-based. Ci vuole esperienza anche se utilizzo modelli algoritmici. L'esperienza però non sempre basta in quanto possono esserci nuove tecnologie, nuovi domini applicativi in cui l'esperienza non c'è.

22.16 Pricing to win

Potrebbe sembrare poco etico e poco orientato al business, ma se non si hanno informazioni, pur di aggiudicare la gara, potrebbe essere un metodo valido. In qualche modo posso poi rivedere lo scope sulla base del prezzo fatto, vincolando lo sviluppo al costo considerato.

22.17 Stima per analogia

Essa si basa sull'esperienza, esperienza però che è memorizzata nei database. È applicabile quando non ci sono metodologie più sofisticate o quando devo avere una stima veloce, semplice e ragionevolmente accurata. La base della stima per analogia è quindi avere informazioni da progetti precedenti. Per aggiornarlo devo avere una size ed uno scopo che sono simili a quello di un progetto di riferimento.

La stima per analogia si basa sull'analogia del nuovo progetto con un solo progetto di riferimento, mentre i modelli algoritmici usano delle equazioni per calcolare i costi sulla base di tutti i progetti che sono sul database. Il nuovo prodotto deve essere similare a quanto sviluppato nel vecchio progetto di riferimento. I metodi di lavoro devono essere similari al progetto di riferimento e deve esserci disponibilità di informazioni dettagliate o persone che hanno lavorato al progetto di riferimento.

La stima è fatta valutando lo sforzo atteso ed il tempo di sviluppo. Si applicano ai costi del progetto di riferimento, dei coefficienti moltiplicativi chiamati modifiers, che tengono in considerazione le differenze percentuali rispetto a certe caratteristiche del mio progetto nuovo con il progetto di riferimento. Per ogni caratteristica in cui c'è differenza tra nuovo progetto e progetto vecchio, si stima la differenza. Il modifier sarà quindi:

- $m_I = 1 + a_i$
- $C = \prod m_i$

- $M = M_{ref} * C$

Bisogna però sapere quali siano le caratteristiche simili e quali siano quelle che differiscono. Tali caratteristiche devono essere **direttamente legate al software**, altrimenti conviene non prenderle in considerazione. Nella pratica **non conviene usare più di 15 modifiers**. Non dovrebbero esserne più di 10 se uno dei modifiers abbia una variazione superiore al 10%.

An example

	estimate	modifier
i System characteristics more complex; about 15%	+15%	1.15
i More menus and screens; about 10%	+10%	1.10
i Structure of data base file less complex; about 5%	-5%	0.95
i More system interfaces; about 20%	+20%	1.20
i Use of DBMS less familiar; about 20%	+20%	1.20
i More powerful development tools; about 10%	-10%	0.90
i Development team with more experience; about 20%	-20%	0.80
i Acceptance testing more rigorous; about 5%	+5%	1.05

$$C = 1.15 \times 1.10 \times 0.95 \times 1.20 \times 1.20 \times 0.90 \times 0.80 \times 1.05 = 1.31$$

In theory no limitations to the number of modifiers that can be used.

In practice no more than 15 modifiers should be used in general and no more than 10 should be used if one of them has a percentage variation greater than 30%

Figura 22.2: Stima per analogia

22.18 Stima dettagliata

Se mi accorgo di avere diverse parti di diversi progetti simili ma di non avere un unico progetto simile, ciò però implica di avere delle stime a livello dell'intero progetto, ma anche delle stime dei singoli moduli. Se è così, posso usare come riferimento dei singoli moduli. In questo caso, devo vedere se un modifier può essere applicato meglio al costo del sistema o al costo del singolo modulo. Non bisogna usare un modifier con valore maggiore di 30% e minore di 5%. Devo inoltre fare attenzione a progetti condotti da diversi team di sviluppo nel tempo, in quanto non ci sarebbe la somiglianza nel team di sviluppo, che però è importante con questo metodo. È necessario usare come modello di riferimento lo sforzo e non il costo, in quanto lo sforzo si riduce in costo sulla base dello stipendio. Quando le differenze sono additive, ovvero devo aggiungere lavoro, non devo considerarlo come modifier ma considerarlo a parte ed aggiungerlo. Nel caso in cui stimi due volte lo stesso modulo, ovvero una volta con l'intero progetto e una volta a parte, prendendo lo stesso modulo da un altro sistema più accurato, devo rimuovere il costo del modulo meno accurato e poi aggiungo il costo di quello più accurato. Ciò potrebbe essere fatto se nel software che sto considerando, un modulo ha una differenza che richiederebbe un modifier troppo grande.

22.19 Modelli algoritmici

Il modello algoritmico più famoso è COCOMO. Il costo viene stimato con una funzione matematica di attributi di prodotto, attributi di progetto e attributi di processo i cui valori sono stimati dal project manager. L'equazione base è:

$$\text{effort} = A * \text{Size}^B * M.$$

A è un coefficiente che dipende dall'organizzazione, B è l'esponente e di solito è maggiore di 1. Esso riflette la diseconomia di scala. M invece è un moltiplicatore che va a considerare caratteristiche ulteriori oltre alla size come prodotto, processo e attributi delle persone. Molti modelli sono simili. Lo sviluppatore di COCOMO ha preso 60 progetti dividendoli in 3 categorie:

- **organic mode:** sono i progetti più semplici, i gestionali. Essi non hanno molti requisiti.

Le operazioni sono:

- MM = $2.4 (\text{KDSI})^{1.05}$
- TDEV = $2.5(\text{MM})^{0.38}$

- **semidetached mode:** sono una via di mezzo tra organic ed embedded. Non sono molto complessi, non avendo le stesse caratteristiche di embedded, **avendo dei requisiti inferiori ma essendo comunque più complessi degli organic**. Le operazioni sono:

- MM = $3.0 (\text{KDSI})^{1.12}$
- TDEV = $2.5(\text{MM})^{0.35}$

- **embedded mode:** il sistema è più complesso. Le operazioni sono:

- MM = $3.6 (\text{KDSI})^{1.20}$
- TDEV = $2.5(\text{MM})^{0.32}$

Il modello COCOMO è stato inoltre sviluppato in 3 forme, **basic** (solo size come equazione), **intermediate** (aggiunta di altri fattori di costo) e **detailed** (aggiunto molto di più). Quello che consente di fare COCOMO è quindi dare delle equazioni sulla stima dello sforzo e del tempo di sviluppo per distribuire lo sforzo tra le varie attività del prodotto. La maggior parte dei progetti studiati da COCOMO erano basati su sviluppo a cascata o **incremental development**. Dalle operazioni possiamo notare che l'esponente aumenta passando da un modello all'altro. Ciò significa che **progetti con maggiore complessità hanno una maggiore diseconomia di scala**. Mediamente, su un progetto very large, ci lavoravano 151 persone. Nella fase di implementazione potevano arrivare ad essere addirittura il doppio.

22.20 Basic COCOMO: Organic mode

Basic COCOMO ed organic mode è applicato alla maggiorparte dei progetti software, utilizzato su progetti di taglia medio-piccola sviluppati da piccoli team. L'esperienza è su progetti simili e si ha la conoscenza di come il prodotto che si sta sviluppando contribuirà agli obiettivi dell'azienda. Si ha un basso communication overhead. Le operazioni sono:

- MM = 2.4 (KDSI)^{1.05}
- TDEV = 2.5(MM)^{0.38}

MM è man-month, KDSI è thousands of delivered source instructions e TDEV sono i mesi. Gli oggetti di tipo organico si dividono in 2000 (small), 8000 (intermediate), 32000(medium) e le 128000(large) linee di codice. Si può notare come all'aumentare della dimensione diminuisce la produttività (diseconomia di scala).

Project Profile	... in generale				
	DSI	MM	DSI/MM	TDEV	FSP
• Small	2.000	5.0	400	4.6	1.1
• Intermediate	8.000	21.3	376	8.0	2.7
• Medium	32.000	91.0	352	14.0	6.5
• Large	128.000	392.0	327	24.0	16.0

Figura 22.3: Basic COCOMO: Organic mode



22.20.1 Limiti di Basic COCOMO

I limiti di Basic COCOMO sono che non considera cost drivers diversi dalle linee di codice. Esso quindi considera solo la size, non tenendo conto del personale, architettura del sistema, complessità, hardware... Il livello di accuratezza è tale da essere solo nelle primissime fasi della definizione del prodotto. Attualmente l'accuratezza è molto bassa.

22.21 Diseconomia di scala

Lo sforzo non è proporzionale alla dimensione del progetto. La crescita è più che lineare poichè se il progetto è più grande, ovvero la dimensione del sistema è più grande, ho bisogno di uno sforzo maggiore nella fase di design per sviluppare specifiche dei moduli che supportano lo sviluppo parallelo di più programmati. Il problema è che durante l'analisi e durante il design, il team non cresce in maniera proporzionale alla dimensione del sistema. Successivamente però si ha bisogno di persone per lo sviluppo, ma il team di analisti non può crescere proporzionalmente al progetto, in quanto design, architettura e analisi dei requisiti è un qualcosa che va fatto in piccoli team.

Il problema non è dover scrivere più roba, ma avendo più moduli si cerca di parallelizzare al massimo lo sviluppo di tali moduli, mettendo più persone nella fase di implementazione. Si mettono più persone in tali fasi per ridurre lo schedule e programmare in meno tempo. È necessario quindi inserire più persone nella fase di programmazione in modo tale che vadano a recuperare il tempo impiegato nelle fasi iniziali di progetto. Il tempo però lo perdo di nuovo nelle fasi di integrazione, in quanto avrà più moduli da integrare. Complessivamente ho quindi uno sforzo maggiore che mi porta ad una minore produttività. La distribuzione dello sforzo segue una curva, chiamata curva di Rayleigh. Essa esprime il numero medio di persone che lavorano ad un progetto in diverse fasi. Sarà necessario fare però un'approssimazione che si adatti al concetto di software, avendo una traslazione. L'idea è che quando comincia avrà un certo numero di persone, così come quando finirà. Il punto di massimo della curva viene fatto coincidere con la metà del tempo di sviluppo. La curva di Rayleigh è una curva continua, ma se io vado a vedere i valori nel database, ottengo una funzione a gradini, in quanto le persone vengono rimosse in numeri discreti, ovvero senza la virgola. Si avrà quindi una smoothed COCOMO, che sarà ben approssimata dalla curva di Rayleigh.

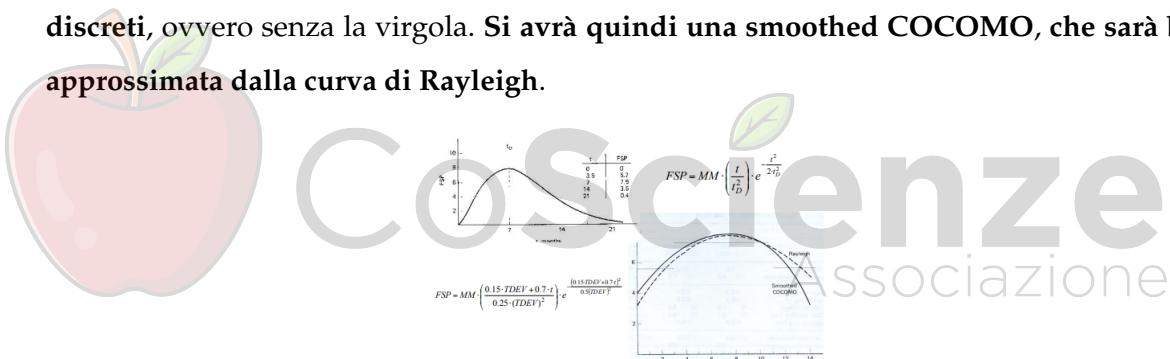


Figura 22.4: Diseconomia di scala

t_D sarà il punto di massimo della curva di Rayleigh, che coincide con la metà del tempo di sviluppo. Il picco di persone si avrà quindi nella parte centrale, che coincide con la parte di implementazione, ovvero a tempo t_D . Si usa una percentuale per indicare il numero di persone medio, poiché alternativamente non riuscirei a confrontare.

22.22 Intermediate COCOMO

Si considerano altri fattori di costo che potessero influenzare. Partendo da 104 e mediante studi sulla loro ridondanza e sull'importanza generale, quest'ultimi sono stati ridotti a 15. Aggiungendo i 15 fattori, si riesce ad ottenere che le stime sono molto più accurate rispetto a Basic Cocomo. Con Intermediate COCOMO si ottiene uno sforzo nominale, che

deve essere poi moltiplicato per il prodotto dei vari fattori. Lo sforzo nominale rappresenta una stima dello sforzo nel caso in cui la dimensione del sistema sia l'unico fattore ad influenzare il costo sia la size. Ogni coefficiente di costo in c_j può assumere un valore di scala crescente da very low a extra high.

22.23 Intermediate COCOMO: cost drivers

I cost drivers di Intermediate COCOMO sono: **affidabilità, selezione del database, affidabilità del prodotto, vincoli sui tempi di esecuzione** (se ho maggiori performances il prodotto mi costa di più), **limitazioni dello storage, capacità del personale...** All'aumentare dell'affidabilità aumentano i costi, mentre all'aumentare delle capacità dell'analista i costi diminuiscono. Ci sono attributi che influenzano positivamente i costi ed altri negativamente. Intermediate COCOMO sono che esso fornisce dei moltiplicatori dello sforzo phase sensitive, in diverse fasi del progetto posso quindi usare diversi moltiplicatori, usando inoltre una detailed three-layer gerarchia, considerando:

- Effetti che tendono a variare con ogni modulo sono trattati solo a livello modulo
- Effetti che variano meno frequentemente sono trattati a livello sottosistema
- Effetti come la dimensione totale del prodotto sono trattati a livello sistema

Non da un vantaggio molto più ampio, ma ha sicuramente dei miglioramenti che va a fornire, fornendo un dettaglio maggiore nelle diverse fasi. Si usa la **sensitivity analysis** è possibile stimare gli effetti dei cambiamenti dei valori dei cost drivers sui costi di sviluppo.

22.24 Problemi dello sviluppo software

I requisiti cambiano continuamente, per cui risulta essere difficile gestire il **cambiamento continuo**. Se il cambiamento è continuo, ovvero i **cambiamenti sono molto frequenti, gestire questo tipo di cambiamenti è ancora più complesso, sia a livello di tempi che di costi**. Cosa non banale è la **gestione dei sistemi legacy** che vanno gestiti.

22.25 Attività di sviluppo

Per venire incontro alle difficoltà, sono state definite le attività nell'ambito dello sviluppo software. Esse sono state classificate in due attività:

- **attività del dominio applicativo:** essa contiene **analisi del problema e progettazione della soluzione**
- **attività del dominio della soluzione:** essa contiene **progettazione a basso livello, implementazione, testing e rilascio**

Tutto ciò ci porta alla **maturità del processo**.

22.26 Maturità del processo

Un processo di sviluppo software è maturo se le **attività sono ben definite ed il management ha qualche tipo di controllo sulla qualità, sul budget e sulla pianificazione del progetto**. Ci interessa sapere se un processo è maturo in quanto, se fosse maturo, è meno probabile che questo processo possa portare al fallimento del progetto. Più è maturo, più è controllabile e più possiamo garantire che il processo non abbia failure, forse. Per misurare la maturità del processo è stato introdotto il **Capability Maturity Model**.

22.27 Livelli del CMM

Modello per la **valutazione della maturità di un processo**. Esso è diviso in livelli:

- **iniziale:** esso è chiamato anche **caotico**. Ogni volta che andiamo a ripetere lo stesso processo, sarà un qualcosa di differente
- **ripetibile:** il processo inizia a seguire step ben precisi, per cui riproducendo lo stesso processo avremo gli stessi step
- **definito:** il **management entra in gioco**
- **gestito:** misure delle attività per cui si hanno dei feedback. Sappiamo cosa stiamo facendo e sappiamo quantificarlo
- **ottimizzato:** ricevendo questi feedback dalle misurazioni, siamo in grado di creare un processo sempre migliore

Esso presenta come vantaggi un **controllo sull'attività**, un **controllo su cosa verrà fuori**, sapendo come possiamo valutare oggettivamente cosa stiamo facendo, per cui **valutazione e pianificazione son ben definite**. Lo svantaggio è che **si rischia di avere un effetto grande fratello**, ovvero una **costante osservazione nell'esecuzione del processo**. Mettere in piedi

questo livello di monitoring e gestione è comunque costoso. Per avere un livello di maturità, noi dovremmo sapere tutto, ma ciò cozza con l'idea di cambiamenti continui. L'ideale modello per la maturità è quindi il modello a cascata per il ciclo di vita del software.

22.28 Modello a cascata

Possiamo tornare indietro da una fase solo quando siamo ancora in quella fase. Non appena si passa alla fase successiva, non è possibile tornare indietro. Per evitare che però il modello a cascata fallisca a causa di una parte mancante nei livelli precedenti, si è pensato di creare un modello a V. Esso è semplice, prevedibile e piace al management in quanto i manager sanno precisamente cosa devono e non devono fare, sanno monitorare il cambiamento. Il problema è che non si ammettono errori. Sbaragliando in una fase non possiamo tornare indietro.

22.29 Modello a V

Ad ogni attività a sinistra, ovvero nella parte di pianificazione, corrisponde un'attività a destra, ovvero nella parte di validazione. Le stesse cose dette per il modello a cascata valgono anche per questo modello.

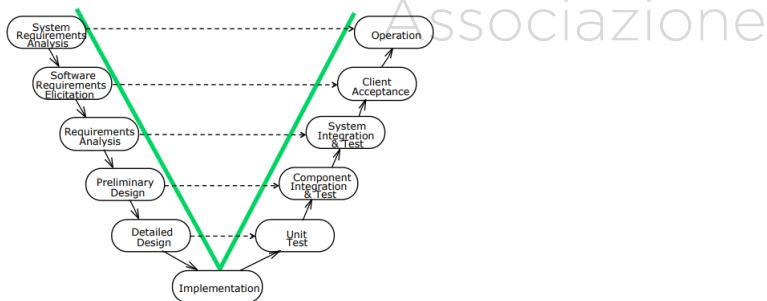


Figura 22.5: Modello a V

22.30 Modello a spirale

Modello che comporta un'attività di tipo iterativo ed incrementale dello sviluppo. Abbiamo dei round all'interno del nostro ciclo di sviluppo, ed ogni round è formato da un modello a cascata con scope ridotto. Questo modello è un passo in avanti ma non è contemplato che qualcosa cambi all'interno del round. La scelta quindi del modello dipende da due fattori:

- **PT = Tempo del Progetto**
- **MTBC = Tempo Medio tra i Cambiamenti**

Se il cambiamento è raro ($MTBC > PT$) allora possiamo usare un modello lineare (Cascata, Modello a V). Se il cambiamento occorre a volte ($MTBC = PT$) possiamo usare un modello Iterativo (Modello a Spirale), mentre se il cambiamento è frequente ($MTBC < PT$) è necessario usare dei modelli agili (eXtreme Programming, Scrum, Kanban).

22.31 Process improvement

I concetti alla base del process improvement sono la qualità del processo di prodotto, analisi e modellazione del processo (dobbiamo comprendere com'è fatto il processo software) e misura. Se non misuriamo non riusciamo poi a capire in maniera quantitativa se gli interventi che abbiamo fatto hanno portato ad un miglioramento.

La prima cosa da fare è quindi comprendere il processo e capire come poter introdurre dei cambiamenti all'interno del processo che possano in qualche modo portare ad un miglioramento su attributi del processo o attributi di qualità del prodotto. Quelli che riguardano la qualità del prodotto mirano molto spesso ad ottenere un prodotto più affidabile e ridurre i difetti. Si potrebbe però mirare ad altri obiettivi di miglioramento.

22.32 Attributi di processo

Esistono diversi attributi di un processo:

- **comprendibilità:** se non riesco a comprendere il processo non posso modificarlo. Esso indica quanto è facile comprendere una definizione di un processo. Un processo è definito se posso modellarlo, se posso rappresentarlo o descriverlo in qualche modo (modelli o non).
- **visibilità:** rappresenta il fatto che io riesca ad individuare l'output di un'attività
- **supportabilità:** supporto che strumenti CASE in ambienti di sviluppo possano dare al processo
- **accettabilità:** il modello di processo viene accettato e utilizzato da parte degli sviluppatori

- **affidabilità:** quanto il processo riesce a recuperare dall'errore e continuare l'esecuzione. Quanto il sistema riesce a recuperare rispetto all'errore
- **robustezza:** se c'è un input che potrebbe creare problemi, si valuta quanto il sistema in presenza di tale input riesca a continuare l'esecuzione
- **manteinability:** può il processo che evolve riflettere i cambiamenti nei requisiti organizzativi o in aspetti di miglioramento che sono stati individuati
- **rapidità:** quanto velocemente il processo può rilasciare il sistema a partire da una data specifica

22.33 Stati della process improvement

La process improvement è similare alla process reengineering di un sistema software, in quanto i passi sono gli stessi:

- **process analysis:** analisi dei processi. Devo capire quale sia la descrizione attuale dei processi. Non quella definita all'interno del manuale. Così come la documentazione non è allineata al codice, anche la documentazione sui processi potrebbe non essere allineata a ciò che effettivamente si fa all'interno di un'azienda. Si modella e si va ad analizzare, possibilmente in maniera quantitativa (si necessitano i dati in questo caso), processi esistenti. Analizzando i dati posso ricostruire modelli di processo. Tecniche sono la mining software repository. I dati sono un'ottima fonte per capire quanto i processi sono eseguiti o se ci sono delle variazioni rispetto a come i processi sono definiti.
- **improvement identification:** Identificare gli aspetti di qualità che posso migliorare, costi o possibili bottlenecks.
- **process change introduction:** devo capire in che modo migliorare un aspetto di un processo o in che modo eliminare un problema. Devo capire quale intervento fare sul processo. Devo modificare il modello di processo e rimuovere i problemi o avere l'obiettivo di migliorare qualcosa
- **process change training:** devo iniziare ad addestrare il personale alla nuova variante del modello di processo. Il modello infatti non sarà subito introdotto ma dovranno essere fatti degli esperimenti per capire se la modifica produce l'effetto desiderato.

- **change tuning:** una volta trovato il modello adatto, posso standardizzare il nuovo processo ed adottarlo

Il **testing** verrà fatto eseguendo istanze del processo. Per testare un software vado ad eseguirlo. Devo quindi istanziare il processo in qualche progetto ed andarlo ad eseguire. L'esecuzione non interessa se sia manuale o automatica

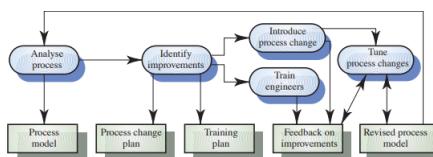


Figura 22.6: Stati della process improvement

22.34 Qualità del processo e del prodotto

Nel caso del software, non è soltanto il processo il principale determinante della qualità del prodotto. Esso ha un ruolo importante ma si dipende anche dalla qualità degli sviluppatori. Fondamentalmente, più è piccolo il progetto più la qualità del personale è importante. Più è grande il progetto, meno è importante la qualità del singolo e maggiore è la qualità del processo, dell'organizzazione... Anche la tecnologia di sviluppo è particolarmente significativa per piccoli progetti. Mettendo uno schedule troppo stretto a qualunque progetto, sarà possibile inficiare la qualità del prodotto.

I principali fattori della qualità del prodotto sono: qualità del processo, qualità delle persone, tecnologie di sviluppo ed i vincoli che posso avere su costi, tempi e schedule.

22.35 Analisi e modellazione

L'analisi è lo studio dei processi esistenti per comprendere le relazioni fra le varie attività, i vari aspetti del processo e confrontarle con gli altri processi. L'analisi dovrebbe portarmi a modellare il processo. Devo quindi produrre un modello astratto del processo, rappresentato tipicamente in maniera grafica. È inoltre necessario individuare i problemi. Devo andare a guardare i modelli pubblicati, ovvero coloro che fanno parte degli standard e sono contenuti nei modelli di qualità aziendali, ma potrebbero esserci di variazioni rispetto a tali modelli. Si potrebbe quindi fare interviste e questionari (ciò viene fatto durante la requirements elicitation). Si potrebbe fare un'analisi etnografica per capire come lavorano

le persone. è possibile utilizzare l'activity diagram con object flow (ovvero vediamo gli oggetti come output ed input) e casi d'uso.

22.36 Activity diagram con object flow

Si ha una dipendenza fra le attività e gli oggetti. Un oggetto dipende da un'attività se quell'oggetto è output di quell'attività. Un'attività dipende da un oggetto se quell'oggetto è input all'attività. Attraverso l'activity diagram con object flow, posso non solo modellare il flusso delle attività, ma anche gli input ed output prodotti da tali attività ed in che modo l'output di un'attività viene usata come input di un'altra. Mediante swim lanes ho l'attore che svolge quell'attività e se l'attività è interattiva, avrò il caso d'uso in cui andrà a descrivere precondizioni e postcondizioni.

22.37 Eccezioni di un processo

Molto spesso ho un modello di processo. Potrebbero però verificarsi delle eccezioni. Esse sono:

- **previste:** esse sono le eccezioni che vengono gestite. Vengono previste per cui so come rispondere ad un verificarsi di tale eccezione
- **non previste:** quando si verifica un errore, devo uscire dall'esecuzione del processo e trovare al momento una strada alternativa, questo perchè non ho gestito l'eccezione in quanto non prevista. Questo diventa un problema quando voglio gestire il processo in maniera automatizzata.

L'idea è di avere una lista di attività ed ogni volta che viene svolta una di esse, la successiva in stato "to do" viene attivata. L'utente che dovrà svolgere l'attività va a vedere nella sua worklist quale sia la prossima attività che deve eseguire. Il motore per gestire questa cosa, deve basarsi sulla rappresentazione dei grafi. Si andrà quindi a definire un'attività mediante le regole "event" "condition" "action". Per cui se termina un'attività e si verifica la condizione, si attiva l'attività successiva. Ognuna di queste attività può essere tradotta in una regola "event" "condition" "action". È possibile prevedere attività che partono allo stesso momento. Esse saranno attività in parallelo.

22.38 Process measurement

è necessario raccogliere i dati in maniera quantitativa. Le misure di processo devono essere utilizzate per valutare i miglioramenti dei processi. Sulla base dei dati posso capire quante volte si verifica una determinata situazione per capire quanto sia problematica la cosa. Si potrebbero inoltre usare informazioni afferenti al tempo richiesto per completare alcune attività.

22.39 Paradigma Goal/Questions/Metrics

Esso consiste di 3 parti:

- **goals:** ci chiediamo quale sia il nostro obiettivo.
- **questions:** servono a ridurre l'incertezza che potrebbe esserci rispetto all'obiettivo generico. Servono a trovare le variabili dipendenti. Questa fase ci guida verso cosa e come vogliamo misurarlo. Le domande non hanno solo l'obiettivo di individuare le variabili dipendenti, che sono quelle che vado a misurare per capire quale sia il livello di usabilità dell'applicazione, ma è necessario anche per capire quali siano gli attributi che devo misurare.
- **metrics:** misure che devono essere collezionate

Il problema è che noi dobbiamo legare gli obiettivi di miglioramento ma non sappiamo come farlo. Iniziamo quindi a farci delle domande su come si possa tradurre l'obiettivo in termini pratici. L'effetto lo devo avere sulla variabile dipendente, ovvero la prima cosa che devo trovare, ma devo anche individuare, mediante questions, quali siano le variabili indipendenti che possono avere un effetto sulle variabili su cui devo agire. Tale paradigma mi collega in qualche modo metriche agli obiettivi, per cui fornisce un modo di capire in che modo certe misure possono aiutarmi ad ottenere ciò che voglio. Esse migliorano l'interpretazione.

22.40 Software Engineering Institute

Istituto di IS, fondato con l'obiettivo di definire tutti gli standard legati allo sviluppo software che aziende fornitrice della difesa americana dovevano rispettare. Per quanto

riguarda la qualità, per valutare la qualità delle organizzazioni, è stato costruita una certificazione specifica, ovvero la CMM. Essa è formata da più livelli, e il livello di certificazione dipende da cosa deve fare l'organizzazione per la difesa americana:

- **initial:** livello incontrollato. In questa fase, a livello di visibilità si sa solo che entra qualcosa ed esce qualcosa.
- **repeatable:** sono state definite ed utilizzate delle procedure di product management. C'è almeno un processo software che viene eseguito, una standardizzazione del prodotto. In questa fase, a livello di visibilità si sa quale sia l'output di ogni attività, sapendo cosa produce ogni attività
- **defined:** ci sono procedure e strategie di gestione del processo che vengono definite ed usate. A questo livello si iniziano a mettere in piedi programmi di misura. In questo caso non so come interpretare i dati ed interpretare le relazioni fra dati. In questa fase, a livello di visibilità si sa quali sono i task che vengono svolti dalle attività, cosa produco
- **managed:** ho delle strategie di gestione della qualità che sono state usate. Ho una gestione della qualità completa insieme ad una quantitative process management. Ho un controllo statistico dei processi. A questo livello sono in grado di interpretare i dati che ho eventualmente raccolti, potendoli mettere in relazione tra loro. So usare la statistica per interpretare i dati raccolti. Se non riesco a gestire in maniera quantitativa i miei processi, non posso fare process improvement, in quanto non posso controllare gli effetti del cambiamento sui miei processi. In questa fase, a livello di visibilità si riesce a capire cosa succede se inserisco qualcosa all'interno del mio processo
- : ho delle strategie di miglioramento dei processi che sono definite ed usate. In questa fase, a livello di visibilità riesco proprio a cambiare il modello di processo, potendo cambiare attività con altre

La differenza con ISO 9000 è che tale standard è statico, mentre CMM è dinamico e fatto a livelli. Tali livelli definiscono le pratiche che un'organizzazione deve avere nei suoi manuali di qualità per poter stare a quel livello. Il vantaggio di gestire la qualità di processi avanzati è che si spende di meno in termini di tempo e costo. La qualità che può sembrare costo in realtà produce guadagno. Se un'organizzazione si trova al livello due di ISO 9000, tipicamente si trova almeno a livello 2 di CMM. Con CMM si vede maggiormente la differenza tra organizzazioni che producono con gli stessi standard.

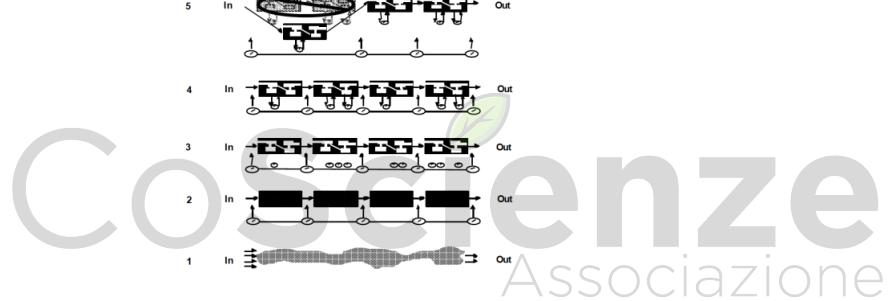
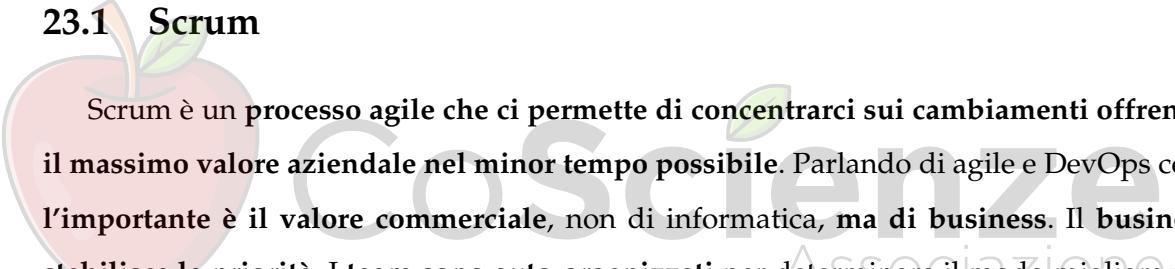


Figura 22.7: Software Engineering Institute

CAPITOLO 23

Modelli agili

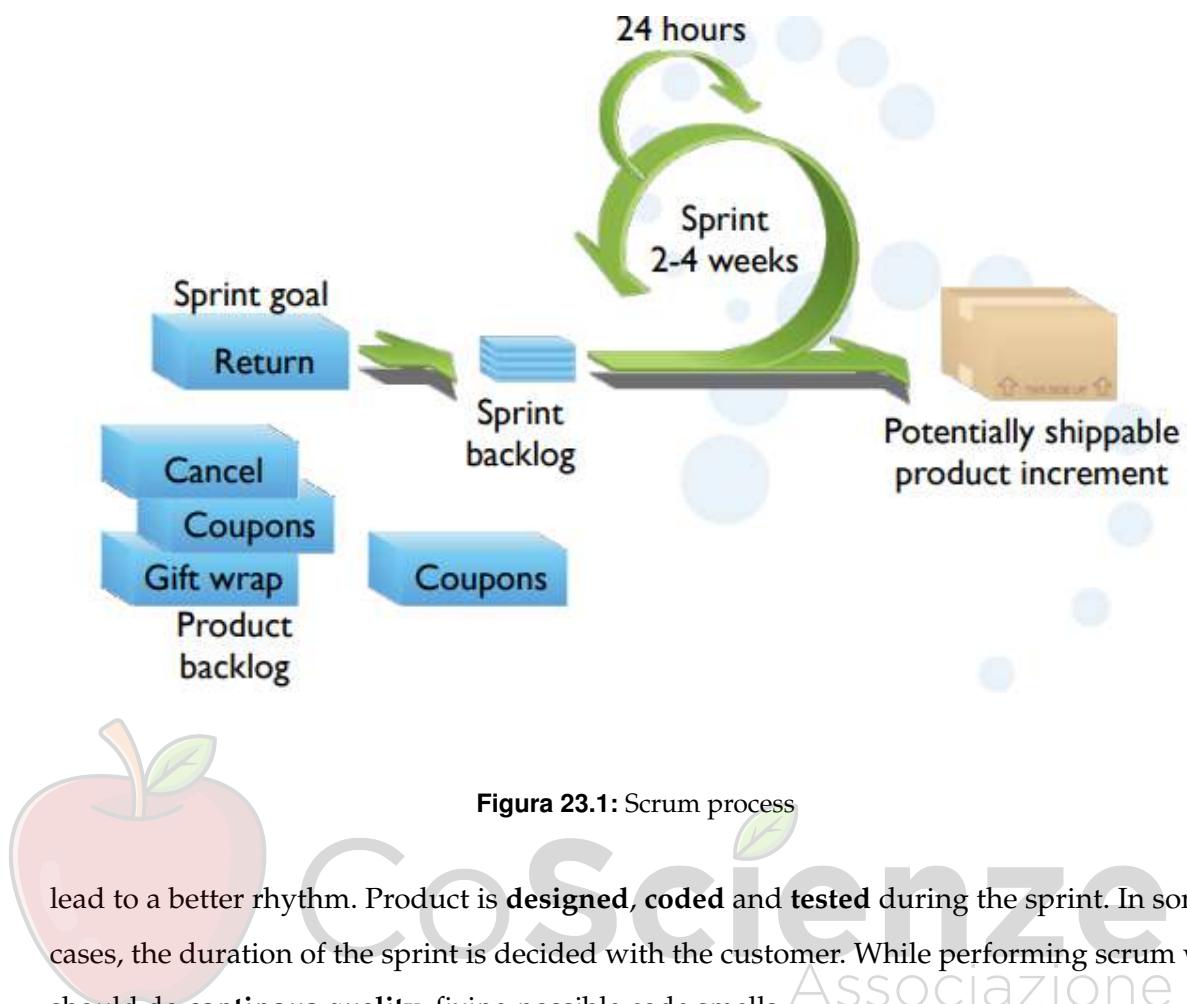
23.1 Scrum



Scrum è un processo agile che ci permette di concentrarci sui cambiamenti offrendo il massimo valore aziendale nel minor tempo possibile. Parlando di agile e DevOps cosa l'importante è il valore commerciale, non di informatica, ma di business. Il business stabilisce le priorità. I team sono auto-organizzati per determinare il modo migliore per offrire il massimo caratteristiche prioritarie. Ogni due settimane o un mese è possibile vendere software realmente funzionante e decidere di rilasciarlo così com'è o continuare a migliorarlo per un altro sprint. In Scrum abbiamo un elenco di requisiti chiamati product backlog. Il team sceglie quali sono i requisiti che desidera implementare e li inserisce nello sprint backlog. Dopo quella scelta, lo sprint (development phase) inizia. Alla fine avremo qualcosa che è spedibile. Negli altri modelli, analisi dei requisiti, progettazione, codice e test dove fasi separate. Con scrum tutte queste fasi sono sovrapposte. Scrum è maggiormente usato quando vogliamo realizzare uno sviluppo commerciale, sviluppo interno o software che richiedono di essere online costantemente.

23.1.1 Sprints

Scrum projects make progress in a series of sprints. Typical duration is 2-4 weeks or a calendar month at most. During the sprint every day there is a meeting, called **daily scrum meeting**. In this period **no one can interfere with the sprint backlog**. A constant duration



lead to a better rhythm. Product is **designed, coded and tested** during the sprint. In some cases, the duration of the sprint is decided with the customer. While performing scrum we should do **continuous quality**, fixing possible code smells.

23.2 Scrum's characteristics

Scrum has:

- **Self organized teams**, called scrum
- **Product progress in a series of 1-4 week sprints**
- Requirements are captured as items in a **list of product backlog**
- **No specific engineering practices prescribed**
- Uses **generative rules to create an agile environment for delivering projects**
- **One of the agile processes**

23.3 Agile Manifesto

The agile manifesto has the idea to put emphasis on some aspects of software development. We should focus on:

- **individuals and iterations over process and tools:** All the socio-technological factors, are enforced by this manifesto
- **working software over comprehensive documentation:** documentation is essential but could be better have a software that is self describable
- **customer collaboration over contract negotiation:** we need the customer to be a part of the group. As soon as we have a problem, we can go to the customer and just ask how to do it
- **responding to change over following a plan:** if there is a change that could change all the requirements in our software, we have to do something. If we continue with the plan that does not contain that change, we will just waste time

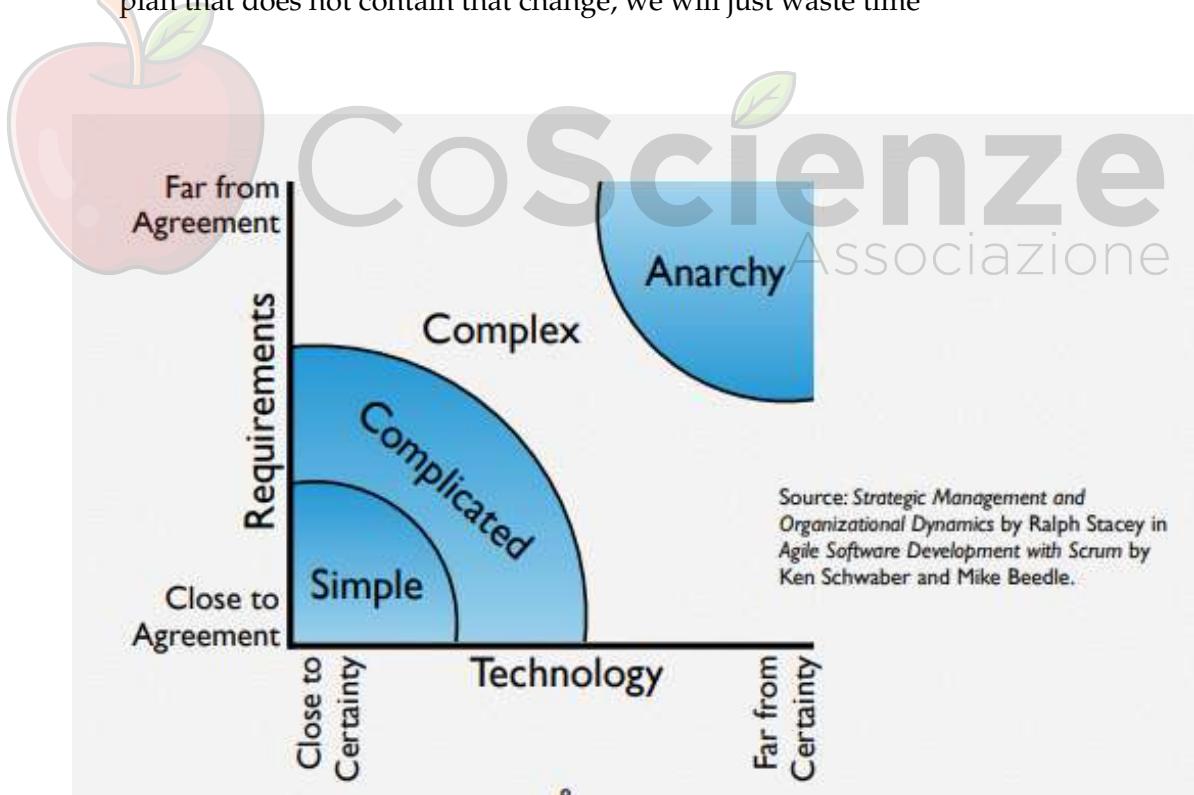


Figura 23.2: standard normalization process

In practice there are 4 kinds of projects that we can develop. There are two main objectives that are in contrast: **requirements** and **technology**. If we have a project for which we know all

requirements and technology very well, the project will be simple. The more requirements and technology are far from certainty, **the software will be complicated, complex arriving to anarchy.**

23.4 Scrum framework

Scrum framework is divided in:

- **Roles:**
 - **Product owner:** defines the features of the product, decides on release date and content. He is **responsible for the profitability of the product**, prioritize features according to market value, adjust features according to market value, adjust features and priority iteration, as needed, accept or reject work results. Sarà il product owner ad accettare o rifiutare il risultato dello sprint
 - **ScrumMaster:** represents the manager of the project. He is responsible for enacting Scrum values and practices, removes impediments and **ensure that the team is fully functional and productive**. Enables close cooperation across all roles and functions, and is a **shield team from external interferences**. If in sequential sprints the team is not capable to produce anything, the scrumMaster will be fired. Non ci sono molte regole in Scrum, ma le poche che ci sono verranno fatte rispettare dallo ScrumMaster, assicurandosi che ognuno rispetti il proprio ruolo.
 - **Team:** Typically 5-9 people. Team is **cross-functional**: programmers, testers, user experience designers... Member should be full-time (may be exceptions) quindi concentrati sul progetto, and teams are self organizing. Membership should change only between sprints.
- **Ceremonies:**
 - **Sprint planning:** A sprint planning is guided by team capacity (in terms of number of team members and effective capacity), product backlog, business conditions, current product and technology defined. During the meeting we have a **sprint prioritization** so we can analyze and evaluate product backlog and select the main sprint goal. With the **sprint planning** the teams meet to understand how to achieve a particular goal, create sprint backlog (tasks) from product backlog items and **estimate the sprint backlog in hours**. In this meeting, also the product owner is

there. The output of this phase are **sprint goal** and **sprint backlog**. For each task we estimate a time, usually max 16 hours. We have to consider high-level design

- **Sprint review:** Team presents what is accomplished during the sprint. Tipically takes the form of a demo of new features or underlying architecture. It is informal and requires 2-hour preparation time rule. There are no slides and the whole team participates
- **Sprint retrospective:** Periodically take a look at **what is and is not working**. During this sprint retrospective, we talk about the process. It is done after every sprint but the idea is to fix the process. The duration is about 15-30 minutes. Whole team partecipates, scrumMaster, product owner, team and possibly customers and others. In this meeting we should understand what **stop doing, start doing or keep doing**.
- **Daily scrum meeting:** It's daily and the duration is about 15 minutes. Usually it is a stand up meeting. It **isn't used for problem solving**. Isn't used for solve a bug but for understand where is the bug and how he effects other components. The whole world is invited. Only team members, ScrumMaster, product owner can talk. It **helps to avoid other unnecessary meetings**. Fare meeting prolungati potrebbe far perdere l'attenzione dei partecipanti

- **Artifacts:**

- **Product backlog:** list of requirements, list of all decided work on the project. It should be developed in a way that requirements are important for customers, interacting with him and should be prioritized by the product owner and reprioritized at the end of each sprint
- **Sprint goal:** short statement of what we are going to do during the sprint
- **Sprint backlog:** represents the work to do during the sprint. Work is never assigned and people in the team should decide what to do. The estimaed work that remains every day is updated. Everyone in the team can change the sprint backlog. Every time there is a new sprint, new work will emerge. If something is unclear we should **provide more time for this item and break it down**. These tasks are meaningfull only if we break them, otherwwise is impossible to understand stuffs. Tools such as **burndown chart** are used to see how is it going. At the start of the sprint we do some estimates for the sprint backlog to be completed, and after

understanding the tasks, estimation can go up (dangerous) and can do down (we are doing a good job). If at the end of the project we are not close to 0, that means that what we had to achieve was not achievable. Il burndown chart è anche utilizzato per dare una rappresentazione visiva dello sprint backlog

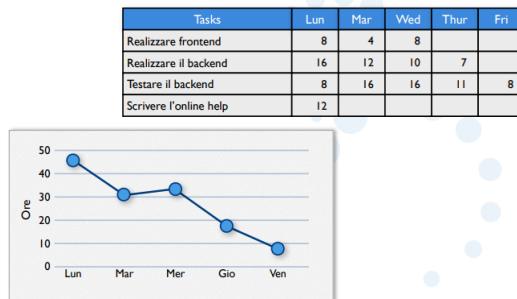


Figura 23.3: Burndown chart

23.5 Scalability

Typically individual teams is composed from 5 to 9 people. **Scalability comes from teams of teams**, and some factors in scaling that could lead to problems are: **type of application, team size, team dispersion** (people all over the world) and **project duration**. In practice we do something called scrum of scrums. Each team has a person that isn't only in the team but also in an upper team. **This person works in a scrum and in a scrum of scrums**, in order to guarantee communication between teams. This hierarchy can go up as much as we want.

23.6 Capacità del team e Focus Factor

La capacità del team indica le ore totali di lavoro di un team. Considerare però solo la capacità del team potrebbe essere un problema poiché **non sempre il team riuscirà a darmi il massimo delle attenzioni**. è quindi necessario considerare il **focus factor**. Il Focus factor è l'abilità del team di restare focalizzato sullo sprint goal senza soccombere ad altre distrazioni. Si aggira in un range tra 0.6 – 0.8. Moltiplicando la capacità totale (ore totali di lavoro) con il focus factor è possibile ottenere la reale capacità del team, rispetto alla quale è possibile fare previsioni, in quanto essa rappresenta in maniera più adeguata le ore effettive di lavoro che ci si può aspettare dal team.

23.7 Stimare le user stories

Le user stories vengono stimate durante lo **spring planning** e la loro priorità viene data dal **product owner** in accordo con il valore dell'azienda. A differenza di altri modelli di sviluppo, per calcolare il **singolo peso** ed il **singolo task**, l'intero team partecipa, adottando una **tecnica chiamata Planning Poker**.

23.8 Planning Poker

Esso è diviso in due fasi:

- **Step 1 Fase Individuale:** Ogni membro del team suppone il numero di giorni in cui secondo lui il task si completa.
- **Step 2 Fase Collettiva:** Si mettono a confronto iterativamente le stime individuali, per tirarne fuori una comune

In alternativa si ha una fase di comunicazione in cui ogni team member da le proprie motivazioni sul perchè abbia dato un determinato peso e sul perchè si richieda un determinato effort per la funzionalità. Un problema è che si guarda alla funzionalità globale e non scomposta nelle sue attività, per cui l'effort potrebbe risultare sbagliata, in quanto dipende inoltre dal team.

23.9 eXtreme Programming

L'idea di **eXtreme Programming** è quello di contrastare i problemi legati al ciclo di vita del software. eXtreme Programming nasce proprio da uno dei firmatari dell'Agile Manifesto. L'idea è che **il cambiamento è una costante ed è necessario trovare una soluzione per poter affrontare tutto ciò**.

eXtreme Programming è un **modello scientifico basato su pratiche già esistenti ma che sono state prese ed adattate in modo che possano essere applicate a qualsiasi contesto** e che **ogni individuo coinvolto in questo contesto abbia il supporto per queste pratiche**. L'idea è quindi di **avere rapidità e concretezza**. Tutto si basa su **cambiamenti frequenti e continui in risposta a feedback continui** e ciò che è importante in eXtreme Programming è il **valore che viene dato alla qualità e al testing**.



Figura 23.4: eXtreme Programming

23.10 Qualità e testing

Fare testing è difficile, ma si dice spesso che se una pratica è difficile è meglio farla più spesso. Questo è l'ideale dietro l'eXtreme Programming. L'idea è che, sapendo quanto sia difficile fare testing è necessario stravolgere il testing, per cui nell'eXtreme Programming è possibile utilizzare la tecnica TDD (Test Driven Development), garantendo un lavoro di maggiore qualità. Chi lavora al software sarà inoltre più motivato. L'idea è che di andare ad aggiustare, man mano che ricevo feedback il sistema, facendo in modo che il costo delle manutenzioni si riduca ed io non arrivi ad avere delle manutenzioni troppo grandi da gestire.

23.11 Quattro valori dell'eXtreme Programming

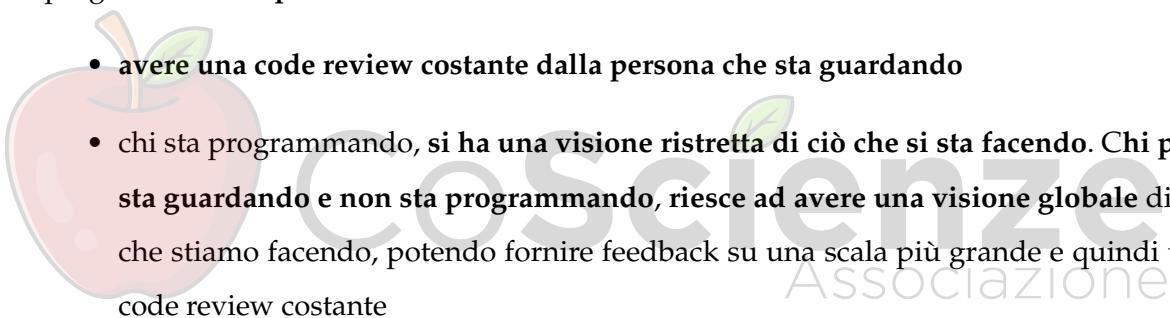
I quattro valori contemplati dall'eXtreme Programming sono:

- **comunicazione:** non è necessario avere documentazione estesa, andando a preferire la comunicazione. La documentazione ci sarà ma sarà essenziale, efficace, diretta. Essa sarà fatta mediante user stories, ovvero requisiti espressi mediante forma comunicativa. Una vera documentazione si ottiene con il testing. Una documentazione così ottenuta sarà sicuramente vera, in quanto il test non mente. La comunicazione prevede anche il task estimation, la pianificazione dell'interazione ma anche il **conceitto del pair programming**
- **semplicità:** più importante sviluppare delle funzionalità che sono semplici, che sono di valore per il mercato, piuttosto che funzionalità complesse. Funzionalità semplici sono più facili da modificare. Se io devo lavorare su qualcosa di semplice, inoltre, anche la comunicazione su quella funzionalità sarà semplice.

- **feedback:** il feedback deve avvenire in tempi diversi. Esso è fondamentale nell'eXtreme Programming. L'idea è di essere costantemente in code review e di ricevere costantemente feedback continui. Il codice viene sempre revisionato. Essa include anche continuous integration ed automated unit test. L'idea è di integrare spesso il sistema
- **courage:** coraggio nel dire se una funzionalità deve essere soppressa o meno. Il fatto che si spenda molto su una funzionalità non è una buona scusa per continuare a usare risorse.

23.12 Pair programming

Due persone sviluppano la stessa funzionalità, ovvero una che sviluppa e l'altra che guarda mentre sviluppo. L'idea è che il fatto di avere una persona che sta aiutando nella programmazione permette di:



- avere una code review costante dalla persona che sta guardando
- chi sta programmando, si ha una visione ristretta di ciò che si sta facendo. Chi però sta guardando e non sta programmando, riesce ad avere una visione globale di ciò che stiamo facendo, potendo fornire feedback su una scala più grande e quindi una code review costante

Pair programming quindi migliora ciò che si sta facendo e la velocizza anche.

23.13 Cinque principi di base dell'eXtreme Programming

I cinque principi di base sono:

- **fornire feedback:** l'idea è che il feedback deve essere veloce, sia tra clienti che tra programmatore.
- **semplicità:** si deve prediligere la funzione più semplice dal punto di vista della soluzione. Non è necessario fare cose complesse.
- **cambiamenti incrementali:** i problemi vanno risolti poco alla volta ed in modo incrementale. Se lasciamo stagnare il problema per molto tempo, sarà difficile risolverlo. Mentre cambiando poco alla volta siamo più in grado di gestire il cambiamento.

- **abbraccia il cambiamento:** L'idea è di **preservare la maggiorparte delle azioni mentre risolviamo la maggiorparte dei problemi**
- **qualità del lavoro:** si deve **preferire un lavoro di qualità**. Ciò dipende anche dalle persone che devono svolgere il lavoro

23.14 Le 12 pratiche dell'eXtreme Programming

Esse sono:

- **whole team:** tutti contribuiscono. Il team deve essere autodeterminato, avendo anche una rappresentanza del cliente. Nel team non esistono dei veri e propri ruoli
- **planning game:** stimare con le user stories
- **piccole release:** il fatto di avere **release piccole e continue** è similare a scrum
- **customer test:** i test sono essenziali ma essi dovrebbero provenire dal cliente e sono quei test utilizzati per far sì che ciò che il cliente ci ha chiesto, sia ciò che gli stiamo dando
- **collective ownership:** se io sto lavorando ad una funzionalità oggi, domani tale funzionalità potrà essere sviluppata da altri. Ciò può essere fatto solo se c'è una buona copertura di test, altrimenti si hanno dei guai
- **coding standard:** si usano degli standard che vanno rispettati. Ciò rende il codice più comprensibile
- **sustainable pace:** l'idea è di **non fare straordinari, lavorare in maniera costante allo stesso ritmo**. Progetti stressanti non producono software di qualità
- **Continous integration:** ogni volta che effettuo una pull request o che completo un'unità di lavoro, essa deve essere integrata con il resto del sistema. Così facendo si mantiene sempre il sistema aggiornato e soprattutto testato, evitando la regressione.
- **pair programming:** due persone che programmano insieme
- **TDD:** sviluppo prima il test e poi vado ad implementare il codice. Può essere complesso all'inizio ma il codice garantisce sempre una coverage pari al 100%. Progettando il test e poi il codice, il codice rispetta le specifiche concretizzate in quanto il test è la specifica

- **Refactoring:** altero la struttura del sistema senza modificarne il comportamento esterno. Si va quindi a migliorare la qualità del codice.
- **Simple design:** il design, essendo bottom up, è molto più semplice. È possibile poi adattarlo mediante design pattern o astrazioni complesse. Eseguendo top down è più complesso, in quanto all'inizio non si tocca nel pratico l'implementazione.

23.15 Problemi dell'eXtreme Programming

Esso è non realistico in quanto incentrato troppo sul programmatore, le specifiche sono dettagliate ma mai scritte, la progettazione vede dopo la fase di testing, richiede refactoring costante (visto come attività superflua) e richiede troppa disponibilità del cliente. Propone inoltre 12 pratiche troppo interdipendenti. L'eXtreme Programming però è talmente versatile da poter essere usato anche da uno sviluppatore singolo.

23.16 Kanban

I problemi attuali nei sistemi fanno riferimento ai bug nel codice in produzione o a problemi legati alla produttività del team. La scarsa produttività è un problema di grande importanza nei contesti aziendali, e ciò può essere legata o ai piani di sprint troppo vaghi, o molti task che i team devono sviluppare nello sprint, o al o alla scarsa identificazione dei colli di bottiglia. Kanban deriva dall'industria per cui è stato successivamente adattato al software.

L'idea è che il metodo della produzione stessa delle componenti fosse direttamente guidato dalla domanda, evitando la sovrapproduzione ma aumentando la produttività dei teams. Kanban è quindi un processo utilizzato dalle aziende per migliorare la produzione. Tale tecnica è stata adattata nel tempo all'information technology e non solo.

Kanban quindi non è un processo prescrittivo (come scrum), ma si ha il permesso di modificare il processo. Ha il permesso di essere diverso cosa che non consente di fare scrum avendo metodologie prefissate, ma la situazione è unica. Essa è ottimizzata per ogni tipo di dominio. Pur essendo uno strumento, Kanban viene utilizzato come una metodologia di sviluppo. Per la gestione di team distribuiti con Kanban vengono in aiuto gli strumenti digitali.

23.17 Funzionamento di Kanban

Kanban è composta da: backlog, in progress, peer review, in test, done e blocked, ma tale lista può variare in base al contesto delle aziende. Sulla bacheca è possibile attaccare delle story card, similari alle user stories. Le story card sono i task che devono essere portate a termine. La tabella rappresenta lo stato del progetto. Kanban è quindi utile per identificare possibili colli di bottiglia ma può limitare il carico di lavoro per ogni fase. Si parte dalle necessità del cliente. I team possono auto organizzarsi intorno ai task e le policy possono migliorare per cambiare i risultati ma soprattutto la soddisfazione del cliente.

Le pratiche di Kanban ci permettono di visualizzare lo stato dei task, limita il "work in progress" ponendo un limite al numero di task in esecuzione, permette di ottenere feedback rapidi dai clienti, rende le policy esplicite e le policy possono collaborativamente cambiare a seconda dei fabbisogni del team e mediante esperimenti.

23.18 Policy e Postit di Kanban

Si usano i postit per ogni task, si scrive in maiuscolo e si segna chi sta svolgendo il task. I post-it dovrebbero essere comprensibili da tutto il team e i task dovrebbero avere una granularità piccola.

23.19 Termini importanti di Kanban

I termini di Kanban sono:

- **Lead Time:** tempo richiesto per completare un task.
- **Cycle Time:** tempo necessario per concludere il task.
- **Throughput:** fa riferimento alla produttività, definita come il carico di lavoro consegnato in un certo time frame.
- **WiP Limit Value Stream:** dipende dal processo di sviluppo.
- **Swarm(ing):** collaborazioni ad un problema.

23.20 Limitare il WiP

Bisogna analizzare quanti task in parallelo i membri di un team possono gestire. Tale verifica deve essere necessariamente fatta con il team, in modo tale da non superare il limite. Sommando i WiP otteniamo il limite totale dell'applicazione. La legge di Little afferma che il leadtime = WiP/throughput, per cui per aumentare il lead time possiamo o aumentare il throughput o diminuire il WiP. Ciò si traduce nell'aumentare il throughput e quindi avere tool più efficienti, lavoro straordinario, aumentare la pressione sul team, avere più incentivi, o diminuire il WiP avendo meno multitask, meno overload e favorire la collaborazione.

23.21 Ottenere ed incoraggiare il feedback

Le cadenze sono riunioni periodiche che può essere un Kanban meeting (giornaliero) o un replenishment meeting (bisettimanale). Il meeting giornaliero è molto simile allo scrum meeting e risponde a tre domande: cosa hai fatto ieri, cosa fai oggi ese hai qualche impedimento. Esso quindi serve a coordinarsi con lo scrum master ma anche per responsabilizzare tutti i membri del team nello svolgere un lavoro e nel dare feedback se si hanno problemi su determinati task

23.21.1 Kanban Meeting

Si analizza la Kanban board da sinistra a destra focalizzandosi sui task bloccati. Si decide come intervenire. La frequenza viene decisa dal team (di solito giornaliero). La durata viene decisa dal team. Di solito è uno standup meeting.

23.21.2 Replenishment Meeting

Serve per pianificare quali task prioritizzare per spostarli dal backlog al WiP. Si dovrebbero prioritizzare i task:

- con valore di business più alto
- maggiormente time critical
- che permettano di ridurre rischi futuri

Se nel team c'è un product owner allora sarà lui a decidere quali task devono essere sviluppati nel periodo successivo

23.22 Ridurre i rischi di ritardi

Segnare sulla card quando avviene un passaggio di stato da una fase all'altra per capire quali task sono in ritardo. Utilizzare metriche e diagrammi per tenere traccia del processo

23.23 Differenza tra Scrum e Kanban

La differenza più importante è che Scrum è un metodo, mentre il Kanban è uno strumento. Kanban costruisce un modello di distribuzione continua in cui i team rilasciano valore non appena sono pronti, mentre Scrum organizza il lavoro in Sprint. La scelta tra i due dipende dalla natura del processo. Kanban offre un approccio su misura, mentre Scrum si basa su regole predefinite.

Kanban è un metodo adattivo, mentre Scrum è prescrittivo. Kanban ha una maggiore concentrazione sulle esigenze dei clienti e gestisce il lavoro... mentre Scrum è basato su esperienza, trasparenza, ispezione ed adattamento. Le cadenze in Kanban sono a livello di team e a livello di servizio mentre Scrum ha una cadenza a lunghezza fissa contenente pianificazione degli sprint, scrum giornaliero, revisione dello sprint e retrospettiva dello sprint. In Kanban si ha responsabile di consegna del servizio e di richiesta del servizio. In Scrum si ha invece titolare del prodotto, scrum master, team di sviluppo e project manager. Le metriche di Kanban si osserva maggiormente tempo di ciclo e throughput, mentre per Scrum si guarda più alla velocità e alla capacità prevista

Su team di maggiore grandezza, essendo sprint a lunghezza fissa, essendoci una pianificazione, Scrum potrebbe dare determinati vantaggi, ma comunque Scrum è una tecnica agile che si basa su team cross-funzionali che non sono grandi.

Tuttavia, Scrum e Kanban possono essere utilizzati insieme, facendo nascere "Scrumban". Il team potrebbe utilizzare Scrum per organizzare il lavoro in sprint e stabilire un ritmo costante di consegna, e utilizzare Kanban per visualizzare il lavoro in corso. La combinazione di queste due metodologie può aiutare il team a massimizzare l'efficienza e la produttività, mantenendo allo stesso tempo un'adeguata flessibilità per adattarsi alle esigenze del progetto e del team. Scrumban non è adatto a tutti i progetti o a tutti i team. Inoltre, dovrebbe essere adattato alle esigenze specifiche di ciascun progetto.

23.24 DevOps

DevOps si basa su **body of knowledge** provenienti da Lean, Theory of Constraints e tutte altre metodologie che vengono o sono correlate a metodologie agili. Il principio chiave su cui DevOps si fonda è quello di **consegnare frequentemente software funzionante**. Lo scopo è di **aumentare la frequenza il più possibile** e, mettendolo in confronto ad approcci tradizionali o approcci a cascata, qui parliamo di **consegnare un prodotto da un paio di settimane a un paio di mesi**, massimizzando il periodo di consegna. DevOps punta a **minimizzare il value stream** e quindi è possibile definire DevOps come un insieme di pratiche volte a ridurre il tempo che intercorre tra il commit di una modifica a un sistema e la modifica nella produzione normale, garantendo al contempo un'elevata qualità. Non è detto che per minimizzare il value stream basta che diminuiamo la qualità andando poi a creare problemi al sistema. Ogni volta che andiamo nell'ambiente di produzione, **dovremo sempre stare attenti alla qualità**.

23.25 Flusso di valore

Il **value Stream** (Flusso di valore) è la sequenza di attività necessarie per progettare, produrre e consegnare un bene o un servizio a un cliente. Dalla fase di pianificazione alla **fase di lancio** usiamo DevOps per rappresentare tutte le attività che fanno parte del ciclo di vita. In DevOps trasformiamo questo concetto e andiamo a definire il **flusso di valore** come il processo necessario per trasformare un'**ipotesi aziendale**, su cui si basa il nostro progetto, in un **servizio abilitato alla tecnologia**, per poi offrilo infine al cliente.

23.26 Lead time

Il **lead time** è il tempo che intercorre tra la creazione di una richiesta e il completamento di un lavoro. Lo scenario ideale di DevOps è **minimizzare parte del lead time** in termini di minuti. Andiamo quindi a **definire il commit**. Andiamo poi a **definire tattiche** che ci permettono di avere un **approval automatico** e che quindi ci permettono di eseguire in un breve periodo un **test automatizzato**. Dalla fase del commit alla fase del testing, deve quindi passare pochissimo tempo. Sarà poi necessario effettuare **manual approval** per poter effettuare **exploratory test** volti a capire se la modifica nell'ambiente di produzione va a rispettare i criteri di accettazione dell'utente, per poi passare **manualmente** alla fase di **production deploy**.

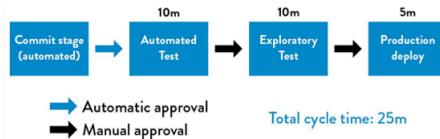


Figura 23.5: Passaggi di DevOps

23.27 Problemi di DevOps

Il problema è che **tutte queste fasi non coinvolgono solo la fase di sviluppo**, non coinvolgono quindi solo lo sviluppatore, ma nello specifico, **la project and deploy**, va a coinvolgere un altro team che DevOps deve affrontare. Il problema è quindi l'interazione tra sviluppatori ed amministratori di sistema, coloro che vanno a gestire il sistema in produzione e quindi gestire la fase di operation.

DevOps va quindi ad interagire con due team particolari che molte volte vanno in contrasto. Il team degli sviluppatori preferisce il cambiamento, preferisce fornire nuove versioni del sistema non interessandosi alla qualità del servizio

L'altro team invece, il team di operation, preferisce avere stabilità del sistema, non punta ad avere frequenti cambiamenti ma piuttosto preferisce ritardare la messa in produzione per poter mantenere una versione del sistema che è stabile e che possa fornire qualità senza andare a fornire evoluzione del sistema.

23.28 Risoluzione del problema di DevOps

DevOps permette di risolvere il problema andando a creare una realtà collaborativa che permette di colmare il divario, invece di mantenere il divario tra la fase di development e quella di operation. Si passa da due team che creano un muro di confusione ad una realtà collaborativa, dove la fase di sviluppo collabora con la fase di Ops, nel senso che tutte le attività svolte dal Dev, sono svolte in funzione della fase di operation. Quello che fanno è preparare l'architettura software che possa essere implementata immediatamente e che possa essere adattata all'ambiente di produzione. Ops deve fare la stessa cosa, ovvero deve osservare il runtime dell'architettura e deve fornire un feedback operativo allo sviluppatore per migliorare continuamente l'architettura.

23.29 Fasi di DevOps

Le fasi che vanno a definire i principi fondamentali di DevOps sono 3:

- **Fast left-to-right flow of work:** a partire dalla fase di development ad arrivare alla fase di Ops, dobbiamo fare in modo che tale operazione sia il più veloce possibile
- **Fast and constant flow of feedback:** a partire dalla fase di Ops, bisogna fare in modo che la fase di Ops aumenti la frequenza di feedback il più possibile in modo da poter migliorare la fase di Dev
- **Continual Learning and Experimentation:** tra Dev ed Ops deve esserci continuo learning. Devono essere applicate delle pratiche di monitoring, delle pratiche di preparazione di comunicazione. Tali fasi devono migliorare nel tempo e quindi andare a creare un livello di maturità per quanto riguarda DevOps

23.30 Garantire DevOps

Per garantire DevOps devo avere quindi:

- **sviluppo e test in un ambiente simile alla produzione:** non possiamo prendere un commit e mandarlo direttamente nell'ambiente di esecuzione, altrimenti potremmo creare dei danni. Per cui quello che possiamo fare è avere un ambiente che sia il più simile possibile alla produzione in modo tale da poter avere che il commit vada ad interagire direttamente e vada a trovare problemi che posso ritrovare nella produzione
- **distribuzione accelerata usando processi e strumenti adeguati:** quello che dobbiamo fare è **automatizzare il processo ed essere veloci**. Quello che DevOps ci fornisce è automatizzare i processi
- **convalida continua del QoS:** questo perchè la qualità deve essere sempre alta
- **collaborazione e feedback continui:** feedback soprattutto per la fase di Operation che deve fornire feedback alla fase di Dev

23.31 Categorie di tecniche del DevOps

Le categorie di tecniche che vado ad utilizzare per **garantire automazione, alta qualità e continous feedback** sono:

- **continous architecting:** è quello di garantire che l'architettura del sistema si evolva nel tempo e che ci permetta di poter considerare sempre gli attributi di qualità e sfruttare il feedback che viene dalla fase di Operation.

- **continuous integration:** andare ad **unire frequentemente copie di lavoro che gli sviluppatori vanno a condividere** in caso di cambiamento in modo da poterli gestire e collegarli nella fase di operation
- **continuous testing:** una volta effettuati cambiamenti ed aggiornamenti del sistema, è necessario effettuare dei test per garantire che ogni distribuzione sia convalidata.

23.32 Continous architecting

L'obiettivo della **continuous architecting** è quello di garantire che l'architettura del sistema si evolva e si adatti continuamente per soddisfare le esigenze aziendali e tecnologiche in un ambiente di sviluppo e operativo in continua evoluzione. Ciò include la creazione di un'architettura modulare, flessibile e scalabile che consenta la distribuzione continua delle diverse versioni del software e delle modifiche nel contesto DevOps. Le **funzionalità chiave della continuous architecting** includono:

- **Microservices:** abbiamo bisogno di servizi che siano il più piccoli possibile, non avendo sistemi monolitici
- **Containerization:** Creazione di container isolati per provare i cambiamenti in ambienti simulati che siano il più simili possibile alla produzione.
- **IaC:** Definizione dell'**infrastruttura del sistema come codice** (Infrastructure as Code). Andiamo a **definire componenti e risorse che interagiscono nella fase di produzione e che noi possiamo andare a riprodurre ogni volta**. Ciò ci garantisce che la **containerization** sia il più simile possibile a ciò che effettivamente è la produzione.
- **Configuration Management Tools:** Configurazione gestione automatica delle **componenti del sistema** (o Orchestrazione), garantendo che l'architettura sia coerente e correttamente configurata in tutti gli ambienti

23.33 Continous integration

La continuous integration (CI) è **una pratica di sviluppo del software che coinvolge l'integrazione regolare e automatica del codice di sviluppo in un repository condiviso**. Quando andiamo ad effettuare un cambiamento, **ciò che vogliamo è che tutte le fasi che riguardano testing, analisi statica del codice, siano effettuate in modo automatico ed il più veloce possibile**. Le **funzionalità chiave della continuous integration** includono:

- **Automazione delle build:** La CI consente di automatizzare il processo di creazione dell'eseguibile, dei pacchetti o del deploy del software. Ciò assicura che il codice sorgente venga compilato in modo coerente e senza errori.
- **Analisi statica del codice:** La CI può eseguire automaticamente analisi statica del codice per identificare potenziali problemi, come violazioni di stile, errori comuni o vulnerabilità di sicurezza.
- **Notifiche e feedback tempestivi:** Facendo un nuovo aggiornamento, anche gli altri sviluppatori devono essere informati della nuova versione del software. Tutti devono essere a conoscenza del nuovo cambiamento nel sistema. In caso di errori o fallimenti, vengono inviate notifiche per consentire agli sviluppatori di risolvere rapidamente i problemi.
- **Reportistica e monitoraggio:** Attraverso metriche di qualità del codice e mediante metriche di processo, possiamo capire come effettivamente migliorare e monitorare DevOps
- **Versionamento del codice:** La CI tiene traccia delle diverse versioni del codice sorgente, consentendo di ripristinare versioni precedenti in caso di necessità

23.34 Continous testing

Avendo effettuato CI, io voglio garantire che ad ogni modifica le fasi del test passino. Consiste nell'esecuzione automatica e continua dei test durante tutto il ciclo di vita del software, in modo da identificare tempestivamente eventuali problemi. Le funzionalità principali del continuous testing includono:

- **Automazione dei test:** Il continuous testing prevede l'automazione dei test, compresi i test unitari, i test di integrazione, i test funzionali, i test di regressione e altri tipi di test. Ciò permette di eseguire i test in modo rapido, ripetibile e affidabile.
- **Integrazione con la pipeline di CI/CD:** Il testing deve essere effettuato subito dopo il commit all'interno del sistema. Ciò ci permette di capire se effettivamente dalla fase di aggiunta del nuovo commit alla fase di deploy è stato aggiunto un errore
- **Esecuzione dei test in ambienti simili alla produzione:** I test vengono eseguiti in ambienti che sono il più possibile simili a quello di produzione, riducendo i rischi di problemi che potrebbero verificarsi solo in ambienti diversi.

- **Test paralleli e distribuiti:** Il continuous testing può essere eseguito in parallelo su più ambienti o su diverse istanze di test per accelerare i tempi di esecuzione e ottenere risultati più rapidi

DevOps ci **permette di usare una serie di tool** che ci permettono di garantire, per ogni fase della pipeline di DevOps, **un supporto**.



CAPITOLO 24

Code smells

24.1 Introduzione: Problemi sul codice

Mediante approccio iterativo, aggiungo delle features che non avevo definito fin dall'inizio, per cui devo rivedere l'architettura per poter accomodare i cambiamenti. Se faccio sviluppo in maniera agile senza però fare questo tipo di refactoring che è preventivo rispetto al degrado architettonico, ottengo un degrado architettonico, ovvero ciò che viene chiamato **smell**. Il refactoring si fa in maniera preventiva o correttiva. Per capire che ci siano dei problemi che richiedono di fare refactoring, è necessario osservare il codice. All'interno di quest'ultimo ci saranno dei sintomi che riguardano problemi di qualità. Tali sintomi prendono il nome di **code smell**.

24.2 Code smell

I code smell sono **sintomi di una cattiva progettazione o di scelte implementative sbagliate**. Una delle principali cause è proprio l'**evoluzione che causa una riduzione della qualità**. Man mano che aggiungo codice ottengo un degrado del design. Tali problemi hanno manifestato, se non risolti, una più bassa produttività e più sforzo degli sviluppatori. Alcuni code smells sono:

- **blob (god class)**: classe che implementa molte responsabilità e ha un largo numero di attributi, operazioni e dipendenze con altre classi. Come conseguenze si hanno costi

di manutenzione più alti, ogni volta che viene modificata la classe **devo modificarne in cascata molte altre**

- **swiss army knife:** molto vicina alla god class. Si parla di una classe che offre molti servizi, una classe che implementa molte interfacce. La differenza tra servizi ed interfacce è la fase in cui essi vengono forniti. I servizi sono un gruppo di operazioni, nel momento in cui vado a specificare la signature delle operazioni, precondizioni e postcondizioni, si ha un'interfaccia
- **shotgun surgery:** ogni volta che si effettua un cambiamento, bisogna fare tanti piccoli cambiamenti a molte classi
- **divergent change:** si ha quando una classe è modificata in diversi modi per diverse ragioni
- **class data should be private:** classe che espone i suoi attributi, violando l'information hiding
- **complex class:** classe con elevata complessità ciclomatica
- **functional decomposition:** classe dove polimorfismo e eredità vengono usati poco per cui si dichiarano molti campi e ci sono pochi metodi, ragionando maggiormente in termini function decomposizion e non object decompositon
- **spaghetti code:** classe che non ha una struttura e che ha molti metodi grossi senza parametri
- **inappropriate intimacy:** due classi che hanno un accoppiamento elevato che non dovrebbero avere
- **lazy class:** classe molto piccola che non fa molto all'interno del sistema
- **long method:** metodi di grande size
- **long parameter list:** metodo con una lunga lista di parametri
- **middle man:** classe che delega il suo lavoro ad altre classi
- **refused request:** classe che eredita funzionalità mai usate
- **speculative generality:** classe astratta che non è necessaria in quanto non viene specializzata in nessun'altra classe

I code smells hanno un **impatto negativo sulla comprensibilità**, aumentando la **change and fault proness, aumentando i costi di manutenzione**.

24.3 Refactoring

Andiamo a cambiare il sistema software in modo tale che non venga alterato il comportamento esterno. L'obiettivo del refactoring è **maggior manutenibilità**. Esistono più di 90 operazioni di refactoring ed alcune di esse sono già implementate di default negli ambienti di sviluppo integrato, altre invece sono più complesse. Alcune operazioni sono:

- **spezzare il codice in parti più semplici:** Ciò può essere fatto a **livello di package**, a livello di **classe** o a **livello di metodi**. Per l'**extract method refactoring** posso usare una tecnica basata su program slicing
- **migliorare il posto in cui il codice deve trovarsi: move method refactoring.** Un metodo che non sta bene in una classe viene spostato in un'altra, move class refactoring da un package all'altro
- **migliorare l'aderenza rispetto l'object orientation:** replace conditional with polymorphism. Se ragiono in maniera procedurale faccio un controllo su un qualcosa che può essere il tipo o sottotipo per poi andare ad effettuare un'operazione piuttosto che un'altra. Ciò può essere fatta in maniera automatica dal linguaggio usando il polimorfismo. Un'altra tecnica è **encapsulate field, pull up field o push down method** Alternativamente si può usare il parallelize method. Sulla **base di un certo parametro, posso invocare un certo metodo**. Non lascio quindi la responsabilità al chiamante ma lascio responsabilità alla classe di decidere, **sulla base del parametro, qual'è l'operazione che deve essere invocata**, il che è più corretto

24.3.1 Extract class refactoring

Applicabile al blob, l'idea è di **dividere il blob in più classi**, ottenendo una classe con minore coesione e globalmente non voglio che aumenti il coupling più del dovuto. L'extract package fa la stessa cosa dell'extract class ma a livello di package. Si parla in questo caso di **promiscuous package**. Per **raggruppare le classi o i package** vado a vedere quali abbiano responsabilità simili, quelli che sono più coese e hanno un maggiore accoppiamento

24.3.2 Move method refactoring

Lo smell che va a curare questo tipo di refactoring si chiama **feature envy**. Esso è uno **smell in cui si evidenzia un coupling elevato a causa di un metodo**, presente in una classe B che però starebbe meglio nella classe A, in quanto si diminuirebbe il coupling e si aumenterebbe la coesione.

24.4 Principi che guidano il refactoring

I principi che guidano il refactoring sono:

- **coesione**: vado a misurare **quanto sono correlate le responsabilità di un modulo**. È desiderabile **un'alta coesione**
- **coupling**: grado con cui ogni modulo è in relazione con altri moduli. È desiderabile **un basso coupling**

24.5 Introduzione di code smells nella pratica

Nella pratica si è osservato che i **code smells nella pratica** si iniziano ad avere non appena si iniziano ad aggiungere le prime funzionalità. Una classe quindi nasce con code smells. Aggiungendo nuove funzionalità non rivedo la progettazione, introducendo delle classi che contengono smells. Le operazioni che vengono fatte quando si introducono code smells sono principalmente **maggiormente quella di enhancement**, seguita da quella di **aggiunta di nuove funzionalità**. Rispetto al workload, i **code smells sono introdotti da chi ha maggiore carico lavorativo** e rispetto all'esperienza sul progetto **chi ha più esperienza**, essendo più carico di lavoro, introdurranno più smells.

24.6 Percezione dei code smells

Pur conoscendoli, la maggioranza degli sviluppatori **non è conscia dei problemi che i code smells potrebbero portare**. L'idea è che **manchino comunque i tool per lavorare sui smells**. Alcuni code smells sono **più evidenti di altri**, come **blob, long method, spaghetti code e complex class**. È stato però dimostrato che, anche quando gli sviluppatori riconoscono che ci sia uno smell, **non fanno niente per risolverlo**. Spesso inoltre si **pospone il refactoring**. I code smells sono **raramente rimossi e anche se vengono rimossi**, non è perché vengano

fatte operazioni di refactoring ma bensì **come side effects**. Ciò significa che si **necessita una maggiore automation**, sia per individuare sia per fare refactoring.

24.7 Refactoring process

Esso è composto da vari step: **dove fare refactoring, come farlo, come garantire che venga preservato il comportamento, applicare il refactoring, assestarsi gli effetti sulla qualità e mantenere la consistenza con gli altri artefatti.**

24.8 Where to refactor

Per individuare da dove partire con il refactoring è **necessario individuare i code smells**. L'idea è che la **maggiorparte degli approcci di detection** individuano i code smells mediante **informazioni che posso estrarre dal codice**. Faccio parsing ed ottengo informazioni di tipo strutturale dal codice. Le metriche che posso tirare fuori sono tutte estratte mediante analisi del codice. Le operazioni di move method e move class, di solito la tecnica che individua lo smell individua anche la soluzione. **Quando individuo un problema di feature envy, io so qual'è la classe in cui c'è lo smell e quindi so anche la classe in cui dovrebbe stare**, ma non è sempre così. In generale però, **la detection dello smell è differente dal refactoring per risolverlo**. Non bastano informazioni strutturali per andare a individuare gli smell. **Alcuni smell infatti sono caratterizzati da come il software evolve nel tempo**. Un esempio è **parallel inheritance** che denota un accoppiamento tra due gerarchie. Per andare ad individuare smell di questo tipo esistono **tool come HIST**.

24.9 DECOR

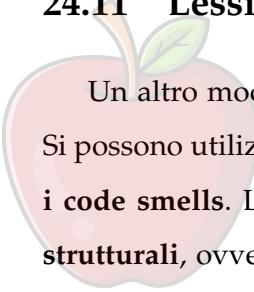
Esso è un **metodo per la specifica e la detection di code smells**. Si parte dalla **descrizione dei dati**, si **effettua domain analysis** e produciamo un **class diagram**, un modello concettuale, una tassonomia. Successivamente, **si fa la specifica**, partendo dal **modello concettuale** ed **ottenendo le rule cards**, andando poi ad implementare gli algoritmi per ottenere degli **algoritmi di detection**.

24.10 HIST

HIST va a vedere i **co-change**, ovvero se due classi vengono modificate insieme. Si osserva l'evento e si osserva se esso viene propagato nel tempo. L'altra tecnica è la frequenza dei cambiamenti. Per **divergent change** se la classe viene cambiata per diversi motivi in diversi momenti, la **detection** viene fatta andando a vedere classi che contengono almeno **due sottoinsiemi** di metodi tali che: tutti i metodi di un insieme cambiano insieme e ogni metodo dell'insieme non cambia con il metodo dell'altro insieme. La soluzione è extract class refactoring

Per i blob, si usa la **frequenza dei cambiamenti**. Se una classe è cambiata molto spesso, probabilmente sarà accoppiata a molte classi e ogni volta che viene cambiata viene cambiata insieme ad altre classi. Ciò potrebbe essere uno dei sintomi. **Non è detto che sia sufficiente per identificare un blob.**

24.11 Lessico del codice e code smells



Un altro modo in cui poter identificare i **code smells** è mediante il lessico del testo. Si possono utilizzare tutte le **informazioni lessicali**, commenti compresi, per identificare i **code smells**. La tecnica è **complementare rispetto a tecniche basate su informazioni strutturali**, ovvero individua smell che non sono individuate da tecniche individuate da informazioni strutturali. Per ottenere informazioni dal codice, **si vanno ad estrarre tutte le parti rilevanti**. Si va ad **estrarre la bag of words**, l'insieme dei termini rilevanti. C'è una lista di **stop words** che potrebbe contenere i **termini del linguaggio che non dovrebbero essere presi in considerazione**, filtrando poi solo i temrini rilevanti.

Un'altra tecnica è la **stop function** in cui si eliminano tutti i termini che hanno un **numero di caratteri minore o uguale ad 3**. L'idea è di **usare sia stop functions che stop words**. Avendo termini composti, posso usare delle regole che si usano tipicamente nel linguaggio di programmazione per andare a fare lo **split** di **termini composti**. Queste sono tutte **tecniche di preprocessing** che vado ad applicare quando vado ad estrarre i termini **dai documenti**. Uso poi tecniche di information retrival per indicizzare le componenti e confrontarle.

Nella tracciabilità, nell'impact analysis, **si usa una componente come query e un insieme di componenti come documenti**, mentre in questo caso quello che viene fatto è confrontare **la somiglianza testuale delle componenti** per capire la somiglianza delle componenti.

Tale tecnica viene usata in **TACO**. Esso prende le componenti, estraе il testo ed applica tutte le tecniche di **separation e removal**, applicando anche **stamming**, ovvero **portare la parola alla radice**. Mediante stamming si eliminano singolari, plurali... Si va poi a calcolare la somiglianza o la dissomiglianza. Con il long method ad esempio, si divide il codice in blocchi e si va a vedere la somiglianza tra i blocchi, estraendo la coesione. **Se la coesione è alta vuol dire che il long method è coeso**, alternativamente non lo è.

Con il **feature envy**, prendo un metodo e calcolo la somiglianza media tra il metodo e tutti i metodi di una classe. Prendo poi tutte le altre classi e vado a calcolare la somiglianza media tra i metodi delle altre classi. Si ha un feature envy quando la **somiglianza con altri metodi di altre classi è maggiore della somiglianza con i metodi della classe stessa**. Si ha quindi un feature envy quando tale differenza è nettamente maggiore. Se due componenti non si chiamano ma **vengono modificate spesso insieme**, significa che c'è comunque **accoppiamento**. In questo caso si parla di **logical coupling**. Esso deriva dalle **informazioni storiche di come i componenti cambiano insieme nel tempo**.

24.12 How to refactor

L'idea è di **combinare varie tecniche per lo smell detection**. L'arrivo degli algoritmi di **machine learning** però non stanno avendo grande successo, in quanto lo **sbilanciamento tra codice buono e code smells è elevato**. Alcune **operazioni di refactoring** possono essere fatte **manualmente**, mentre quelle più complesse richiedono necessariamente dei tool. **Possono esserci dei refactoring mutualmente esclusivi** e ogni volta che viene fatto refactoring, è **necessario ritestare tutto**. Possono essere utilizzate due classi:

- **critical pair analysis:** va a vedere i conflitti, ovvero quali sono i **refactoring mutualmente esclusivi**
- **sequential dependency analysis:** definisce quali sono i **refactoring propedeutici ad altri refactoring**.

I **refactoring semplici** possono quindi essere fatti **manualmente**, quelli più complessi richiedono algoritmi ad hoc, potendosi basare sulle facility dell'ambiente di sviluppo per poter effettuare operazioni più semplici richieste all'interno di operazioni di refactoring. Oltre a capire i refactoring incompatibili, devo capire quali sono gli esclusivi e quali devono essere effettuati dopo un altro. Alcuni refactoring migliorano quindi alcuni aspetti mentre altri ne migliorano altri.

24.13 Supporto automatico

Supponendo di avere un blob semplice, i possibili modi in cui può essere diviso in diverse classi è un numero abbastanza elevato da richiedere comunque un effort considerevole. è quindi necessario avere tool semi automatici che vanno a raccomandare operazioni di refactoring. Per capire quali sono le soluzioni giuste rispetto alla risoluzione di un code smell

24.14 Tipi di relazioni nei code smell

Ci sono vari tipi di relazione che possono essere individuati:

- **strutturali (analisi statica)**: essa è composta da chiamata ai metodi, metodi che accedono a stessi attributi, relazioni di ereditarietà ed original design. Potrebbe esserci infatti un motivo per cui è stata progettata una certa cosa in un certo modo, per cui non posso ignorarlo. Sono facili da catturare in quanto si usa parsing, static analysis, ma non sono sufficienti. Non è necessario avere tutto il codice che sia funzionale o compilabile
- **dinamiche (analisi dinamica)**: vanno a vedere le dipendenze che però vado a catturare durante l'esecuzione. Sono più precise delle statiche ma sono più difficili da eseguire e devo progettare bene i casi di test, altrimenti potrei avere una bassa recall. Ho bisogno del codice completo che deve essere compilabile
- **semantiche**: l'idea è che usando termini simili vuol dire che i componenti sono simili. Sono facili da catturare. L'assunzione è che i developers usano consistentemente i termini nei commenti. Se viene meno questa osservazione non è possibile usare questa tecnica
- **storiche**: si parla di logical coupling tra le componenti e che contribuiscono a integrare funzionalità simili. Si deve avere una disponibilità delle versioni precedenti

Le più facili da avere sono la strutturale e la semantica. Le più facili da estrarre sono a loro volta strutturale e semantica. La migliore che cattura l'accoppiamento è la semantica. Mediamente comprensibili sono la strutturale e la dinamica.

24.15 Scelta dell'algoritmo per generare la soluzione

La scelta dell'algoritmo per la generazione della soluzione di refactoring, può essere fatta su diversi algoritmi:

- **algoritmi di partizionamento:** è necessario scegliere quali siano le partizioni. Iterativamente poi si va a riorganizzare la somiglianza tra le partizioni, andando a riorganizzare i cluster
- **algoritmi gerarchici:** non si definisce il numero di cluster, per cui possono essere individuate molte partizioni.
- **graph-theory based:** uso una rappresentazione a forma di grafo e applico poi un algoritmo che mi consente di tagliare il grafo in sottografi. Nel grafo ci saranno degli archi e questi archi avranno un peso sulla base di certe misure che ho considerato. Un'altra tecnica è di tagliare il grafo usando dei filtri. Sulla base di questi filtri potrei arrivare ad un grafo non fortemente connesso ma formato da grafi isolati
- **algoritmi euristici:** ho delle regole e sulla base delle regole posso andare ad effettuare lo split
- **algoritmi search based:** il problema si formula vedendo ogni individuo come un sottoinsieme di classi. Devo finire le operazioni di trasformazione, selezione...

24.16 Guarantee behaviour preservation

è necessario garantire che le operazioni fatte preservino il comportamento del sistema e poi dobbiamo applicare refactoring. Per garantire che il comportamento sia preservato è possibile controllare manualmente. Si può usare un qualcosa di più automatizzato che richiede formal proof, dovendo garantire che l'operazione sia corretta. è necessario assicurarsi quindi che il comportamento dopo il refactoring sia lo stesso di prima del refactoring

24.17 Apply the refactoring & Access its effects on quality

è importante preservare il comportamento perchè mediante studi si è visto come il 15% delle operazioni di refactoring introduce bugs, mentre l'introduzione di bugs è al 40% per le operazioni complesse. Spesso il refactoring è vista come un'attività che non necessariamente deve preservare il comportamento o viene vista come un'operazione che

introduce bugs e quindi evitata, ma ciò non va bene. è possibile però introdurre errori anche con le operazioni di refactoring automatizzate. L'unico modo per garantire il refactoring, assicurandoci di non aver introdotto errori, è mediante regression testing.

24.18 Consisrently modify other artifacts

è poi necessario modificare in maniera consistente gli altri artefatti. Facendo refactoring al codice, i casi di test che testano il componente cambiato, potrebbero avere necessità di cambiare, potendo propagare cambiamenti fino agli artefatti di alto livello. è quindi importante mantenere la tracciabilità.

24.19 Effetti collaterali del refactoring

A volte non basta che sia garantito il comportamento funzionale, ma potrebbero esserci requisti funzionali che andrebbero rispettati. è necessario imporre quindi dei vincoli:

- **Temporal constraints:** per verificare che la sequenza di operazioni venga fatta con lo stesso ordine o che un'operazione venga fatta prima dell'altra
- **resource constraints:** il software dopo il refactoring non deve richiedere più risorse di quelle che chiedeva prima
- **safety constraints:**

Tali requisiti non funzionali posso ancora verificarli con testing, ma sia per requisiti funzionali che non funzionali posso definire dei vincoli che devono essere rispettati andandoli a verificare. Si hanno quindi due modi, testing o verification. Mediante la verification ho dei vincoli che esprimo in maniera formale e che poi vado a verificare che dopo il refactoring tali vincoli siano rispettati. La verifica formale è verifica statica, la dimostrazione di verifica di proprietà che vengono estratte dal codice.

24.20 Impatto sulla qualità

Facendo refactoring, nello specifico extract class refactoring mi aspetto che la coesione aumenti e non dovrebbe molto aumentare molto l'accoppiamento globale. Anzi, l'accoppiamento della singola classe nel confronto delle altre classi dovrebbe essere minore dell'accoppiamento del blob con le altre classi. Alcuni refactoring migliorano alcuni aspetti,

altri ne migliorano altri. è possibile inoltre misurare l'impatto sulla qualità interna e poi tale impatto avrà un effetto sulla qualità esterna. è possibile usare formalismi come le assertions per verificare che i vincoli vengano rispettati e che quindi sia preservato il comportamento. Le metriche vengono usate per misurare la qualità prima e dopo il refactoring. Usare metodi informali per la verifiche formali è costoso e difficile, per cui si preferisce fare testing.

24.21 Graph trasformation

Alcune tecniche applicano trasformazioni su grafi. Esse possono anche essere delle trasformazioni su grafi formali, ovvero applico dei pattern a dei grafi. Se verifico un pattern posso quindi una certa trasformazione che implementa un certo refactoring. Per quella trasformazione posso anche aver dimostrato che preserva il comportamento. Se applico la trasformazione, allora la trasformazione è corretta. Le relazioni vengono espresse in qualche modo sui grafi come nodi o archi. I grafi posso usarli per implementare algoritmi che suggeriscono refactoring e per implementare algoritmi che effettuano il refactoring sul grafo. Le tecniche di graph transformations sono difficili da realizzare. Se invece si ha l'euristica che suggerisce il refactoring, implementando poi nel codice, in qualche modo ti supporta.

24.22 ARIES

ARIES è un tool che suggerisce il refactoring e crea una lista di metodi che dovrebbero stare nelle diverse classi. Potevo poi spostare i metodi da una classe all'altra, raffinando il refactoring. Una volta fatto ciò, il tool faceva un refactoring automatico, andando a **splittare il codice e a crearne di nuovo, non applicando però graph trasformation**. Per i metodi usava le facilities dell'ambiente di sviluppo integrato, per le variabili d'istanza veniva usata un'euristica in cui la variabile d'istanza andava nella classe in cui c'erano i metodi che più frequentemente la utilizzavano. Tali metodi continuano ad **accedere alla variabile d'istanza chiamandola per nome**, ma vengono aggiunti anche getter e setter per le variabili in modo che se un metodo che accedeva a quelle variabili era finito in un'altra classe, allora esso poteva usare i getter e setter invece di usare direttamente la variabile d'istanza. Tutto questo automaticamente, usando le facilities dell'ambiente di sviluppo integrato. è possibile variare la soglia di coesione aumentando il numero di classi generate.

24.23 System mutation

Utile per fare **tuning** dei valori. Prendo delle classi del sistema originale e le fondo. Creo nuove classi come unioni delle precedenti. Ho un **sistema mutato** ed applico il refactoring a **queste classi**. Applico poi la **MOJO effect** che indica il **numero di operazioni necessarie per trasformare una classe in un'altra**. Meno operazioni mi servono più alta è la misura. Se le classi in partenza però non sono buone il sistema ottenuto non sarà buono, per cui l'obiettivo è **fondere classi di alta qualità**. Ogni valore usato per il tuning avrà un peso basato sull'importanza di tale variabile. Da un'insieme di move method refactoring è facile presumere che ci sia una extract class refactoring



CAPITOLO 25

Test di regressione

25.1 Introduzione: test e refactoring

Noi abbiamo modificato qualcosa e **dobbiamo testare la cosa che abbiamo aggiunto**, dovendo inoltre verificare che **ciò che non abbiamo toccato continui a funzionare correttamente**. Il problema è che il testing di regressione costa. **Eseguire tutti i casi di test che abbiamo è dispendioso**. Possiamo ridurre il numero di casi di test, e ciò può essere affrontato mediante **test suite minimization, test case selection e test case prioritization**.

25.2 Testing di regressione

Consiste nel **ritestare parti di programma non modificate**, utile a verificare che le **parti nuove modificate non abbiano introdotto problemi nelle parti non modificate**. Si testa prima la nuova componente e poi si applica regression testing. I due tipi di test sono quindi:

- **testing funzionale su ciò che abbiamo modificato**
- **testing di regressione su ciò che non abbiamo modificato.** Si riandranno ad eseguire i test case relativi alle vecchie funzionalità e si **andrà poi a confrontare il comportamento del task prima e dopo dell'operazione di refactoring o manutenzione**. Vogliamo che i comportamenti siano gli stessi

Il problema è che **potrebbero esserci delle dipendenze tra ciò che abbiamo modificato e ciò che non abbiamo modificato**. Tali dipendenze però non dovrebbero esserci. Se il

nostro sistema non funziona su codice non modificato a causa di una modifica fatta su un'altra componente, vuol dire che c'erano delle dipendenze tra le classi non considerate nella fase di impact analysis. è possibile che tale componente sia stata inserita nel CIS ma successivamente si sia scelto di non modificarla, o in alternativa la componente non è stata proprio inserita nel CIS. Non potendo sapere se sia stata fatta bene o meno l'impact analysis è necessario effettuare testing.

25.3 Test all

Strategia che permette di rieseguire tutti i casi di test validi realizzati nelle versioni precedenti del programma P. Un testing di regressione migliore di questo non c'è, il problema però è il costo. L'idea è quindi di cercare strategie che evitino di testare di nuovo tutto. Si vuole ridurre il numero di casi di test da eseguire.

25.4 Selective testing

Alternativa al test all che permette di selezionare solo alcuni casi di test e non tutti. Si usano le stesse tecniche dell'impact analysis in quanto si andrà ad eseguire testing di regressione solo delle componenti che hanno dipendenze in qualche modo con i componenti modificati. Se ho sbagliato l'impact analysis però, sbaglierò anche questa fase. Ciò potrebbe essere un problema.

25.5 Tecniche di ottimizzazione

Tecniche che non vanno a considerare le relazioni tra componenti. Ottimizzare vuol dire eseguire un sottoinsieme della test suite, sottoinsieme che possa rappresentare un ottimo rispetto gli obiettivi. Si vanno in questo modo ad eliminare casi di test ridondanti, posso andare ad eseguire tra i test ridondanti quelli che mi costano di meno, considerando inoltre che l'ordine con cui si eseguono i casi di test può influire sui tempi necessari per identificare i difetti. Anche l'ordinamento può essere importante.

25.6 Test suite minimization

Trovare la test suite di dimensione minima in grado di avere lo stesso grado di "copertura" della test suite di partenza, rispetto ad un dato criterio di copertura. Questo perchè la

test suite con il tempo cresce per cui è necessario definirne un sottoinsieme:

- **input:** insieme di casi di test $T = \{t_1, \dots, t_n\}$ ed un programma P
- **problema:** Trovare un sottoinsieme minimale T^k incluso o uguale T che testi/esegua le stesse parti di codice eseguite da T , ovvero abbia la stessa copertura di T . I criteri di copertura possono essere **branch coverage, statement coverage...**

25.7 Variante della test suite minimization

Una variante della test suite minimization è:

- **Input:** insieme di test case $T = \{t_1, \dots, t_n\}$, insieme di elementi del codice da coprire (statement, ecc.), matrice di copertura. Ogni elemento della matrice assume valore 1 se l'elemento E è coperto dal caso di test, 0 altrimenti
- **Problema:** Trovare un sottoinsieme minimale T^k incluso o uguale T che testi/esegua le stesse parti di codice eseguite da T . Bisogna quindi trovare un vettore X dove x_i è 1 se t_i è selezionato, 0 altrimenti, tale che la somma degli x_i è minima, ovvero il vettore contiene il numero minimo di elementi, e $\text{cov}(X)$ ovvero la coverage dell'array X , calcolata come, $M * X^T$, deve essere maggiore o uguale di 1. Risultato di $\text{cov}(X)$ andrà quindi ad indicare, per ogni elemento, quanti sono i casi di test che coprono quell'elemento. Ogni elemento dovrà quindi essere coperto almeno una volta

Il problema di Test Suite Minimization è **NP-completo**. L'hitting set, ha un insieme universo ed una collezione di sottoinsiemi come input ed il problema è trovare il più piccolo insieme tale che l'unione dei sottoinsiemi sia P . Esso è quindi **identico al problema della test suite minimization, cambiando solo il dominio del problema**. Potrebbe comunque esserci che a parità di copertura, non riesco a rilevare un malfunzionamento.

25.7.1 Minimization: algoritmo greedy

L'algoritmo greedy parte da un insieme vuoto di test case ed incrementalmente aggiunge il test case che copre il maggior numero di elementi del programma da testare.

25.7.2 Minimization: additional greedy

L'algoritmo **AdditionalGreedy** parte da un insieme vuoto di test case ed incrementalmente aggiunge il test case con la massima copertura “addizionale” rispetto alle istruzioni

Greedy (P, T)	
(1)	$C \leftarrow \emptyset$ insieme di elementi coperti di P
(2)	$S \leftarrow \emptyset$ insieme TC selezionati
(3)	$W \leftarrow T$ Worklist inizialmente contenente T
(4)	repeat
(5)	$T_j \leftarrow$ test case in W con massima copertura
(6)	$C \leftarrow C \cup Cov(T_j)$
(7)	Rimuovi T_j da W
	$S \leftarrow S \cup \{T_j\}$
(8)	until $C = Cov(T)$

Figura 25.1: algoritmo greedy

già coperte. Nel caso di prima invece, non si prendevano in considerazione le istruzioni già coperte. Tale algoritmo potrebbe prendere un numero minore di casi di test rispetto al greedy normale, ma ciò dipende dal problema in sé. Gli algoritmi greedy potranno dare infatti la soluzione sub-optimal e non la soluzione ottima.

AdditionalGreedy (P, T)	
(1)	$C \leftarrow \emptyset$ insieme di elementi coperti di P
(2)	$S \leftarrow \emptyset$ insieme TC selezionati
(3)	$W \leftarrow T$ Worklist inizialmente contenente T
(4)	repeat
(5)	$T_j \leftarrow$ test case in W con massimo $ Cov(T_j) - C $
(6)	$C \leftarrow C \cup Cov(T_j)$
(7)	Rimuovi T_j da W
	$S \leftarrow S \cup \{T_j\}$
(8)	until $C = Cov(T)$

Figura 25.2: additional greedy

25.8 Regression test selection

Versione modificata del selective testing, in cui si va a selezionare un sottoinsieme della test suite sulla base delle informazioni relative al programma, alla versione modificata e alla test suite. Nella versione di problemi di ottimizzazione, il regression test selection viene visto come un problema a due obiettivi. Essa è nota anche come test case selection. La struttura è:

- **input:** insieme di test case $T = t_1, \dots, t_n$, programma P
- **problema:** Trovare un sottoinsieme T^k incluso o uguale T che (i) massimizzi il numero di elementi in P testati/coperti da T^k e (ii) minimizzi il costo di esecuzione e copertura del codice e costo di esecuzione sono però problemi contrastanti

25.8.1 Test case selection

La formulazione formale del test case selection è:

- Input: insieme di test case, insieme di elementi del codice da coprire (statements, ecc.), matrice di copertura M in cui il valore di ogni elemento è 1 se l'elemento E_i è

coperto da t_i , 0 altrimenti, un insieme di informazioni sul costo/tempo di esecuzione dei test case. L'insieme C avrà quindi informazioni c_i sul costo dell'esecuzione di t_i

- Problema: trovare un vettore X dove $x_i = 1$ se t_i è selezionato, 0 altrimenti, tale che si vuole minimizzare il costo degli elementi selezionati e massimizzare la coverage. La formula è: $\min \text{cost}(X) = \sum_{i=1}^n (x_i * c_i)$ $\max \text{cov}(X) = M * X^T$ Vogliamo avere un bilanciamento tra minimo costo e massima coverage. Anche il test case selection è NP-compelto, in quanto similare al weighted set cover. Nel problema di prima, il costo era uguale per tutti. Nella scelta del caso di test quindi non si faceva caso al costo. Qui invece si considera il costo, cercando di massimizzare la copertura

25.8.2 Weighted Set Cover

Il problema è così definito:

- Input: insieme universo, collezione di sottoinsiemi di P , ed insieme di costi
- Problema: trovare l'insieme T^* di costo minimo tale che l'unione di t_i appartenenti a $T^* = P$

L'hitting set può essere ricondotto al weighted set cover, mettendo come peso 1 a tutti quanti

25.8.3 Selection: algoritmo greedy

L'algoritmo greedy parte da un insieme vuoto di test case ed incrementalmente aggiunge il test case con la massima copertura per unità di tempo

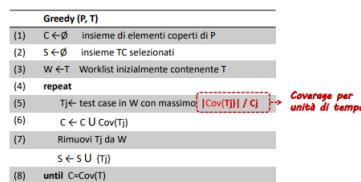


Figura 25.3: Selection: algoritmo greedy

25.8.4 Selection: additional greedy

L'algoritmo AdditionalGreedy parte da un insieme vuoto di test case ed incrementalmente aggiunge il test case con la massima copertura "addizionale" per unità di tempo rispetto alle istruzioni già coperte. Negli algoritmi greedy sarà necessario avere un criterio

di terminazione che mi dice quando fermarmi una volta trovata la soluzione. In questo caso il criterio di terminazione è la copertura massima. Anche con gli algoritmi genetici avrà un confronto e anche in quel caso non troverò necessariamente la copertura massima, ma posso scegliere una copertura minore che però costa di meno. Essa sarà comunque ottima sul fronte di Pareto. In questo caso invece, con l'algoritmo greedy, bisogna raggiungere la copertura massima

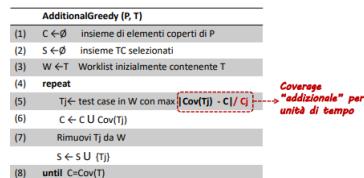


Figura 25.4: Selection: additional greedy

25.9 Test case prioritization

I test case sono ordinati in modo tale da eseguire per prima quelli che hanno maggiore probabilità di testare parti di codice con maggiore probabilità ad avere difetti. Vado a creare un ordinamento mettendo prima quelli che hanno più probabilità di individuare gli errori. Il problema è definito come:

- **input:** insieme di casi di test $T = t_1, \dots, t_n$ e programma P
- **problema:** Trovare un ordinamento ottimale dei test case T^* , in modo tale da eseguire per prima i test case che testano parti di codice difettose

L'idea è che l'ordine con cui siano eseguiti i casi di test influisce sulla percentuale di fault identificati. Un buon ordinamento rivela fault prima di altri, oppure rilevano per prima fault più critici o rilevano per prima fault di parti di codice più critiche.

25.9.1 Funzione APFD

La funzione che ci fa capire quanto è buono l'ordinamento rispetto ai fault individuati è l'APFD (Average Percentage of Faults Detected). Si andrà a prediligere il valore dell'APFD alto.

$APFD = 1 - (TF_1 + \dots + TF_n / n * m) + (1/2n)$ dove n è il numero di casi di test, m è il numero di fault e TF_i è la posizione del primo test case che rileva il fault F_i . Maggiore sarà la posizione in cui un fault viene scoperto, minore sarà il valore di APFD, per cui si cerca di

diminuire la posizione in cui la maggior parte dei casi di test è scoperto, dando precedenza ai casi di test che hanno più probabilità di identificare casi di test. Anche nel migliore dei casi, il valore non potrà mai arrivare ad essere 1.

25.9.2 Problemi dell'APFD e soluzioni

Il numero di fault identificati da un test case è però sconosciuto fin quando il test case non viene eseguito ed il suo output non viene confrontato. Si usano quindi euristiche per derivare quali test case hanno maggiore probabilità di rilevare un bug. Coprendo di più è più probabile che si vadano ad individuare più fault, mentre mi posso basare sulla copertura dei past fault, andando a vedere nel passato quali fault un caso ha individuato un caso di test. Così come per la copertura dei fault, anche per confrontare la copertura degli elementi di codice si può usare una funzione, chiamata APEC (Average Percentage of Element Coverage).

$APEC = 1 - (\sum_{i=1}^m TE_i / n * m) + (1/2n)$, dove n è il numero di casi di test, m è il numero di fault e TE_i è la posizione del primo test case che copre l'elemento E_i .

25.9.3 Definizione formale di test case prioritization

La definizione formale è:

- **Input:** insieme di test case $T = t_1, \dots, t_n$, insieme di elementi del codice da coprire (statements, ecc.), matrice di copertura del codice M in cui il valore di ogni elemento è 1 se l'elemento E_i è coperto da t_i , 0 altrimenti, **matrice di copertura dei fault passati PF** dove, **ogni elemento è 1 se F_i è coperto da t_i , 0 altrimenti**
- **Problema:** trovare un ordinamento T^* di T tale che: $\max \text{CodeCoverage}(T^*) = APEC(T^*)$ e $\max \text{PastFaultCoverage}(T^*) = APFD_{past}(T^*)$. è possibile considerare anche il costo di esecuzione c_i di ciascun caso di test t_i . Esso sarà un costo da minimizzare. APEC e APFD dovranno però essere cambiati. Nel caso in cui il costo di ogni t_i sia uguale ad 1, allora il risultati di APEC e APFD saranno uguali, costi considerati o non considerati.

25.9.4 Prioritization: Algoritmo greedy

Anche in questo caso l'algoritmo è NP-completo. Si va quindi a prendere il caso di test con massima copertura e massimi past fault, ciò però andando a minimizzare costo. Se due obiettivi sono da massimizzare si moltiplicano, se sono da massimizzare e minimizzare allora essi si dividono

AdditionalGreedy (P, T)	
(1)	$C \leftarrow \emptyset$ insieme di elementi coperti di P
(2)	$PastFault \leftarrow \emptyset$ insieme fault coperti
(3)	$S \leftarrow \emptyset$ insieme TC selezionati
(4)	$W \leftarrow T$ Worklist inizialmente contenente T
(5)	repeat
(6)	$T_j \leftarrow$ test case in W con $\max Cov(T_j) - Cov * PF(T_j) - PastFault / C $
(7)	$C \leftarrow C \cup Cov(T_j)$
(8)	$PastFault \leftarrow PastFault \cup PF(T_j)$
(9)	Rimuovi T_j da W
(10)	$S \leftarrow S \cup \{T_j\}$
(11)	until $C = Cov(T)$

Figura 25.5: Prioritization: Algoritmo greedy

25.10 Conclusioni: testing di ottimizzazione

In conclusione:

- **Test Suite Minimization:** minimizzare numero di test case
- **Test Case Selection:** massimizzare la copertura del codice e minimizzare il costo di esecuzione
- **Test Case Prioritization:** massimizzare la copertura del codice (APEC), minimizzare il costo di esecuzione e massimizzare APFD dei fault passati



25.11 Algoritmi genetici e software engineering

Esiste un settore della software engineering chiamato SBSE che si basa sull'uso di algoritmi search based per problemi di software engineering. I problemi di IS possono essere formulati come problemi di ottimizzazione, soprattutto in ambito di testing e di refactoring.

25.12 Backtracking & Branch and bound

L'algoritmo di back tracking è una tecnica algoritmica che ricerca le soluzioni esplorando lo spazio di ricerca. Le esplora tutte andando a confrontare la fitness function. Essa si basa su una ricerca esaustiva dello spazio di soluzioni. Nel caso del branch and bound, come nel caso del backtracking, costruisco incrementalmente la soluzione, andando di volta in volta a vedere la fitness function. Se la fitness fuunction fino a quel momento è maggiore della soluzione ottima, è inutile continuare la ricerca, per cui si usano delle euristiche per terminare prima la ricerca. Ciò elimina un po' di soluzioni, in quanto ogni soluzione parziale può esplodere in più soluzioni finali, avendo un vantaggio maggiore rispetto al backtracking.

L'euristica può essere anche un po' più sofisticata, potendo fermarmi quando capisco che non posso arrivare ad una soluzione migliore. **Sia backtracking e branch and bound mi dà la soluzione ottima, essendo però esponenziale e quindi non trattabile.** Branch and bound **restringe lo spazio di ricerca**, mi dà la soluzione ottima ma è esponenziale. **Esistono però algoritmi di ricerca che hanno complessità polinomiale** poichè restringono lo spazio di ricerca delle soluzioni allo spazio lineare rispetto alle dimensioni del problema. **Non c'è garanzia però che la soluzione sia quella ottima.** L'idea è quella di **adottare strategie per la soluzione sub-ottima**

25.13 Exploration ed exploitation

Dal momento in cui non abbiamo un budget infinito, **non si può fare una ricerca esaustiva**, è necessario usare delle euristiche per andare ad avvicinarci alla soluzione ottima. I problemi di ricerca cercano quindi di **trovare un bilanciamento tra:**

- **ricerca locale (exploitation):** si cerca di sfruttare la soluzione esistente per migliorarla. Il problema dell'exploitation è che **se abbiamo imboccato una strada che ci porta ad un ottimo locale, raggiunto l'ottimo locale non riusciamo a sfruttare la soluzione per migliorarla.** Non riusciamo a più migliorare la soluzione, dovendo usare il concetto di ricerca locale. Gli algoritmi come hill climbing fanno solo exploitation
- **ricerca globale (exploration):** essa mira ad esplorare lo spazio di ricerca il più possibile, massimizzando la diversificazione delle soluzioni

Il **search budget** è il numero massimo di fitness evaluations, ovvero la valutazione della funzione di fitness, compatibili con il tempo massimo di esecuzione dell'algoritmo. È un numero di iterazioni, numero di volte in cui si può ripetere l'algoritmo

25.14 Hill climbing

Algoritmo che **cerca la soluzione migliore a partire dalla soluzione che ha trovato in quel momento.** Il problema degli algoritmi come Hill climbing è **legato al fatto che non riescono a spostarsi migliorando la soluzione, finendo in un ottimo locale.** Questo perchè sono algoritmi di ricerca locali, problema in parte mitigato dagli algoritmi di ricerca globale. Il fatto che sono **algoritmi di ricerca globale, non impedisce di finire in un ottimo locale,** pur avendo una minore probabilità. Esso, insieme al **simulated annealing, sono algoritmi di**

ricerca in quanto esplorano lo spazio delle soluzioni. I problemi sono quindi che potrebbero finire in un ottimo locale non potendo più uscire. Esso si sposta in una certa direzione per cercare di migliorare la soluzione. Ho il valore massimo corrente fino a quel momento, potrei cercare nei dintorni (se è una variante dell'Hill climbing). **Se mi sposto ma non miglioro la soluzione, sono rimasto in un ottimo locale** per cui l'algoritmo si ferma o dopo un certo numero di iterazioni o in base al search budget.

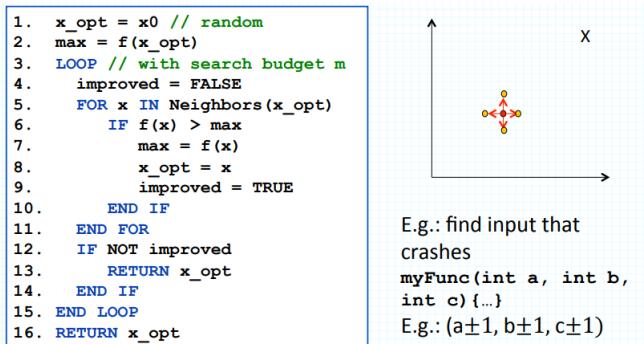


Figura 25.6: Hill climbing

Ci sono alcune varianti che cercano di aggiungere more exploration quando la local search non è efficace, spostandosi ad un altro punto dello spazio.

25.15 Problemi di ottimizzazione

I problemi di ottimizzazione possono essere di due tipi:

- **monobiettivo:** c'è solo una funzione da ottimizzare.
- **multiobiettivo:** ci sono più funzioni obiettivo, non necessariamente tutte da massimizzare o minimizzare

25.16 SBSE

I problemi di software engineering possono essere posti in modo tale da richiamare problemi di ottimizzazione. Alcuni esempi sono:

- Qual è il più piccolo sottoinsieme delle test suite che copre tutte le istruzioni del codice?
- Qual è la migliore allocazione delle risorse per un particolare progetto software?
- Qual è la migliore decomposizione del sistema che massimizza la coesione delle classi?

Per formulare tali problemi è **necessario definire una funzione di fitness che riesca a rappresentare il problema di ottimizzazione**, e usare algoritmi di intelligenza artificiale per trovare l'ottimo

25.17 Algoritmi genetici e SE

Immaginando di avere lo **spazio di esecuzione molto ampio**, volendo evitare di usare algoritmi di ricerca locale, **con gli algoritmi genetici possiamo prendere un insieme molto piccolo delle soluzioni e usarlo come base per esplorare lo spazio di ricerca**. L'idea alla base degli algoritmi genetici è la selezione naturale, ovvero **nella sua evoluzione, la popolazione migliora**. Per definire quindi un problema di SE mediante algoritmi genetici è necessario definire:

- **funzione da ottimizzare**
- **come rappresentare la soluzione al problema.**
- sulla base di come rappresentiamo la soluzione, è **necessario definire gli operatori genetici che verranno applicati** durante l'evoluzione per modificare e far evolvere la popolazione
- tipo di **operatore di selezione**. La selezione degli elementi potrebbe essere fatta in base al valore di fitness, prendendo quelli con valore maggiore
- **operatore della combinazione e successivamente mutazione**. Se non ci fosse mutazione genetica, la popolazione non si riuscirebbe ad adattare ad un contesto. Attraverso la combinazione, crossover, si potranno creare dei figli per la nuova popolazione e quindi la nuova iterazione

25.18 No free lunch teoreme (NFL)

Dato un search budget m , la probabilità media di ottenere una near-optimal f^* usando un algoritmo A è la stessa della probabilità di trovare un near-optimal usando un altro arbitrariamente algoritmo B. Se noi consideriamo due algoritmi qualunque di ricerca e consideriamo la loro applicazione su tutte le possibili istanze di un problema, mediamente esse avranno la stessa probabilità di trovare la soluzione ottima. Se la probabilità media è la stessa, vuol dire che per uno stesso problema avrà delle istanze del problema su cui

l'algoritmo A va meglio e su altre in cui va meglio la probabilità B. Non c'è quindi un algoritmo che va meglio di un altro. **Il fatto che puoi trovare istanze su cui l'algoritmo va sempre meglio di un altro è una conseguenza del NFL.**

25.19 Fasi degli algoritmi genetici

Le fasi degli algoritmi genetici sono:

- **definizione della popolazione iniziale:** La popolazione iniziale è formata da un insieme di soluzioni generate casualmente. La rappresentazione può essere un **cromosoma a lunghezza fissa**, per cui ogni cromosoma ha un gene, che può essere un bit, un intero o un valore generico. Avendo un cromosoma a lunghezza fissa, potrebbe capitare che non tutti gli elementi all'interno del cromosoma contribuiscono alla funzione di fitness. Tale porzione viene chiamata **intron**. Essi non contribuiscono al fenotipo, ovvero ciò che mi permette di calcolare la funzione di fitness. Quando si sceglie una rappresentazione a lunghezza fissa, e l'input può avere una lunghezza variabile, la parte finale dell'input non serve nel calcolo della funzione di fitness. In alternativa, possiamo avere cromosomi a lunghezza variabile.
- **selezione:** I migliori individui (soluzioni) sono selezionati per la riproduzione. Una tecnica di selezione è la **roulette wheel selection** in cui quelli con funzione di fitness maggiore avranno più probabilità di accoppiarsi.
- **crossover:** Riproduzione (ricombinazione genetica degli individui migliori). Ogni crossover può produrre uno o due figli. Per mantenere la grandezza della popolazione fissa, si possono applicare tecniche di elitismo, andando a mantenere parte della popolazione precedente, ovvero quelli con funzione di fitness maggiore, scartando l'altra restante popolazione precedente e sostituendola con parte o l'intera popolazione nuova, questo sempre in base a quanti figli si generano. Bisogna cercare di mantenere il meglio della generazione precedente ed i meglio di ciò che ho generato, mantenendo costante la popolazione. Il crossover può essere single point, two point crossover...
- **mutazione:** mutazione casuale di alcuni geni
- **Si va a vedere se si è soddisfatti della soluzione.** In caso non si è soddisfatti si ritorna alla fase di selezione. **Non è detto che ad ogni iterazione la popolazione venga**

migliorata, per cui anche il degrado della popolazione potrebbe essere un criterio di terminazione, insieme al tempo impiegato.

Per ogni fase esistono diverse variazioni. Ad esempio, successivamente al crossover, posso prendere la popolazione, riprodurla ma prendere solo i migliori di ciò che viene prodotto. Nella selezione invece, si possono applicare tecniche differenti. La **scelta delle tecniche dipende dal problema che si sta affrontando**. L'iterazione continua fin quando si ha search budget.

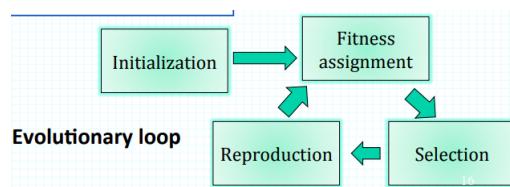


Figura 25.7: Algoritmi genetici

25.20 GAs, exploration & exploitation

Gli algoritmi genetici hanno dei parametri (size della popolazione, tipo di selezione, crossover e mutazione...) e anche il design di crossover/mutazione determinano il trade off tra exploration ed exploitation:

- mutazione del valore con un valore vicino, two-parent crossover, bassa frequenza con cui faccio mutation, elitismo... sto applicando una local search, per cui c'è molta exploitation e molta exploration
- disruptive mutation (cambio il valore con uno molto lontano) più mutazione faccio più esploro, per cui con alta mutazione, multi-parent crossover e ordered selection, faccio global search, per cui faccio maggiore exploration

25.21 Tipi di selezione

Alcuni tipi di selezioni sono:

- truncation selection: prendo i migliori m individui e li faccio riprodurre
- roulette wheel selection: la probabilità di selezionare un individuo si basa sulla sua funzione di fitness. Un individuo può quindi essere selezionato più volte. In questo caso bisogna stare attenti a costruire la popolazione successiva. Con la roulette wheel

selection ho anche exploration. Se io non mantengo i migliori ma li perdo e **accetto di prendere lo stesso individuo più volte, allora sto facendo anche exploration.** Per prendere sempre i migliore dovrei dagli probabilità 1 in un intervallo da 0 a 1, sistemando poi gli altri sulla base della distanza tra i valori delle fitness functions. è quindi una trasformazione di scala. In questo caso gli elementi sono proporzionali alla funzione di fitness, differenziandosi dall'ordered selection

- **torunament selection:** T individui vengono randomicamente selezionati e comparati. Il vincitore del torneo viene inserito nella nuova popolazione. Rispetto roulette wheel selection ciò mi garantisce che il più forte lo prendo sempre.
- **ordered selection:** la probabilità di selezionare un individuo è proporzionale alla sua posizione nella lista di tutti gli individui, posti in base alla funzione di fitness crescente. Preso un elemento in questo caso esso non può essere più preso
- **elitismo:** salvare una copia degli individui migliori e li passo. Agli altri applico altre tecniche. In questo caso gli elementi sono proporzionali alla posizione.

25.22 Automated input generation

Problema monobiettivo. Quando abbiamo un problema di coverage, abbiamo dei target da coprire. Nel caso di coverage target dobbiamo coprire dei cammini, branch coverage dei branch, statement coverage gli statement... Quello che possiamo fare è utilizzare GAs per generare automaticamente test cases capaci di effettuare statement, branch, target coverage... L'idea di base è quindi quella di definire una popolazione iniziale, come ad esempio gli input da dare al programma. è poi necessario definire la fitness function. La final test suite conterrà tutti i cromosomi che sono stati trovati e coperti. Essa però potrebbe contenere ridondanze che possono essere eliminate con un post-processing

25.23 Test data generation: Fictness function

Supponendo di voler coprire branch o statement avendo un target t (che può essere nodo, istruzione o arco). Una delle fitness function più usate è quella che va a calcolare la fitness function come somma di approach level e branch distance:

- **approach level($P(x)$, t):** dato un input x, vediamo la sua execution trace ottenuta eseguendo il programma P con input x. L'approach level è il minimo numero di control

nodes tra uno statement che è stato eseguito ed il coverage target. Si parte quindi dal predicato che sia stato eseguito più vicino al target. L'input che si avvicina di più è quindi quello che va ad eseguire il predicato da cui il target dipende direttamente. Quello che mi indica che non ho eseguito il target è la branch distance.

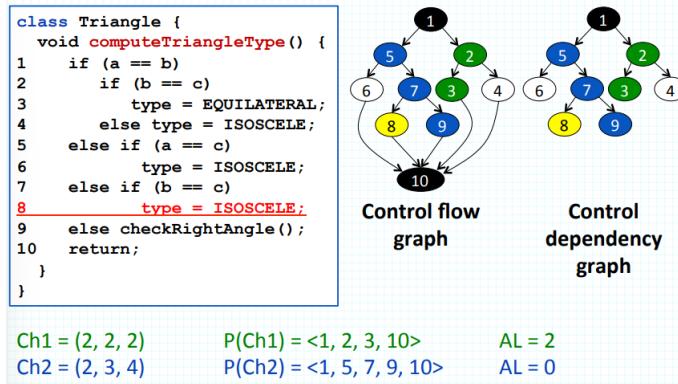


Figura 25.8: Approach level

- **branch distance(P(x), t):** Dato il primo control node dove l'esecuzione diverge dal target t, il predicato del nodo è convertito in una distanza, dal prendere il branch desiderato, normalizzata tra 0 ed 1. Dobbiamo vedere quanto è distante il predicato dall'assumere il valore booleano che mi consente di eseguire il target. Con la branch distance c'è un formulario da considerare. Si usa β in $(d/(d+\beta))$ per far diventare d un valore diverso da 0. Supponendo d=0, per far diventare 0 un valore diverso da 0 mi basta un qualsiasi valore maggiore di 0. Si utilizza invece k nel caso in cui bisogna far diventare ad esempio a = b. Si utilizza k per generalizzare la funzione

Condition c = composite predicate	Distance BD(c) = d / (d + 1)
!p	Negation is propagated inside p
p & q	d = d(p) + d(q)
p q	d = min(d(p), d(q))
p XOR q = p & !q !p & q	d = min(d(p)+d(!q), d(!p)+d(q))

Alternative normalizations of d:
 $BD(c) = 1 - \alpha^d$
 $BD(c) = d / (d + \beta)$
 with $\alpha > 1$ and $\beta > 0$

Figura 25.9: Branch distance

La fitness function sarà quindi data da approach level e branch distance. La distanza del target sarà 0 quando sia approach level che branch distance saranno 0. Si avrà un valore di fitness function pari ad infinito, andando ad utilizzare approach level e branch distance nel

Condition $c = \text{atomic predicate}$	Distance $BD(c) = d / (d + 1)$
a	$d = \{0 \text{ if } a == \text{true}; K \text{ otherwise}\}$
$\neg a$	$d = \{K \text{ if } a == \text{true}; 0 \text{ otherwise}\}$
$a == b$	$d = \{0 \text{ if } a == b; \text{abs}(a - b) + K \text{ otherwise}\}$
$a != b$	$d = \{0 \text{ if } a != b; K \text{ otherwise}\}$
$a < b$	$d = \{0 \text{ if } a < b; a - b + K \text{ otherwise}\}$
$a \leq b$	$d = \{0 \text{ if } a \leq b; a - b + K \text{ otherwise}\}$
$a > b$	$d = \{0 \text{ if } a > b; b - a + K \text{ otherwise}\}$
$a \geq b$	$d = \{0 \text{ if } a \geq b; b - a + K \text{ otherwise}\}$

Figura 25.10: Branch distance

caso in cui l'**input fornito non sia valido per il problema** che stiamo considerando. Un nodo a è dipendente sul controllo da b se b ha due archi uscenti, e seguendo un arco viene sempre eseguito a, mentre eseguendo l'altro a non viene eseguito.

25.24 Collateral coverage

Eseguendo un target, ho eseguito un cammino, per cui non ho eseguito solo quel target, poichè per target si intende sempre arco o branch. Quando però un target è coperto esso viene rimosso dalla lista di target che devono essere coperti. Se io riesco a coprire tutti i target, usando al più un valore pari il numero di McCabe, riuscirò a coprire tutti i target. Quando però un target è coperto accidentalmente, esso è rimosso dalla lista dei target da coprire e il relativo test case viene salvato. Se però i test cases coinvolti nella collateral coverage non sono detected e stored, è possibile dimostrare come un random testing sia migliore del search based testing.

25.25 Problemi multi obiettivo (MOP)

Problemi nel quale bisogna ottimizzare più funzioni contemporaneamente. In molti problemi della vita reale, le funzioni da ottimizzare sono in conflitto tra loro. Ottimizzando x rispetto ad una unica funzione si traduce spesso in risultati inaccettabili rispetto alle restanti funzioni. Una perfetta soluzione multi-obiettivo che ottimizza allo stesso tempo ogni funzione obiettivo è spesso impossibile.

Nel caso di problemi di ottimizzazione del testing si avrà un problema multiobiettivo, in quanto si vuole minimizzare il numero di elementi e massimizzare la coverage. Con l'algoritmo greedy continuiamo fin quando non si arriva alla coverage massima. Nel caso degli algoritmi genetici abbiamo una popolazione di soluzioni, che deve evolvere andando ad ottimizzare due fitness function, una minimizzare il numero di casi di test e l'altra è

massimizzare la coverage. A meno che non si condensino le due funzioni in una, non si riesce ad applicare un algoritmo monobiettivo. In questo caso si potrebbe avere una funzione di fitness unica rappresentata dalla **coverage/ numero casi di test**. Facendo così, ho trasformato un problema multiobiettivo in uno monobiettivo, potendo applicare il problema multiobiettivo. La soluzione del problema potrebbe essere una sul fronte di Pareto, ma non è detto. Mi potrei trovare nel caso in cui la mia soluzione è dominata da qualcun'altra nel problema multiobiettivo.

Problema nei MOP è che le funzioni da ottimizzare possono essere in conflitto tra di loro, per cui bisogna trovare un trade off. Provando quindi ad ottimizzare per una funzione si ha che le altre funzioni non sono ottimizzate dovendo quindi arrivare ad un compromesso. Problemi che possono essere semplici nella loro versione monobiettivo diventano complesse nella versione multiobiettivo.

25.26 Concetto di dominanza

Una soluzione A domina un'altra soluzione B se per tutti i valori della funzione obiettivo, la A è migliore della B. Potrebbe però essere che la A abbia un valore maggiore per una funzione di fitness e B per un'altra. In tal caso nessuna delle due domina l'altra e potrebbero essere entrambe due soluzioni ottime, trovandosi sul fronte di Pareto. Andando a condensare da multiobiettivo a monobiettivo, non è detto che la soluzione che vado ad individuare si trovi sul fronte di Pareto, rispetto all'insieme delle soluzioni ottime che sta sul fronte di Pareto. La soluzione infatti potrebbe essere dominata da altre, in quanto l'algoritmo genetico della funzione monobiettivo si è concentrato sul massimizzare il rapporto e non sul trovare un equilibrio tra le due funzioni obiettivo. La soluzione però sarà comunque sub-ottima. Usando un algoritmo multiobiettivo e trovando una soluzione sul fronte di Pareto, non saprei quale soluzione scegliere, diventando poi soggettiva, in quanto ci sono più soluzioni ottime e non solo una.

25.27 Fronte di Pareto

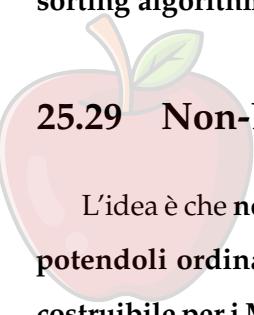
Per sapere quale soluzione scegliere nei MOP, potendo non avere una soluzione che vada meglio in tutte le funzioni obiettivo, entra in gioco il **concetto di Fronte o Ottimo di Pareto**:

- Dati due vettori di soluzioni x e $y \in R^n$, diremo che x domina y secondo Pareto ($x \leq_p y$) se e solo se:

- il valore della funzione obiettivo $f_i(x) \leq f_i(y)$ per ogni $i=1,2,\dots,m$
- $f_j(x) < f_j(y)$ per almeno un indice $j \in 1, \dots, m$ Si intende quindi che per **almeno una funzione il valore deve essere minore, per le altre non deve essere maggiore.**
- Un vettore di soluzioni $x^* \in \text{RegioneAmmissibile}$ è un ottimo di Pareto per le funzioni f_1, \dots, f_m se non esiste in $\text{RegioneAmmissibile}$ tale che $f_i(x) \leq_p f_i(x^*)$. Si sta quindi intendendo che per ogni i , non esiste nessun'altra soluzione y tale che $f_i(y) < f_i(x^*)$.

25.28 GAs e MOPs

Per i problemi monobiettivo gli individui migliori, supponendo un problema di minimo, sono quelli che **minimizzano il valore della funzione**. Nei MOP devono minimizzare più funzioni contemporaneamente, per cui è molto difficile trovare un ottimo assoluto, ovvero un qualcosa migliore di tutti gli altri per tutte le funzioni obiettivo. Nel caso di MOPs per sapere gli elementi da selezionare si utilizza un algoritmo chiamato **non-dominated sorting algorithm** (NSGA).



25.29 Non-Dominated Sorting Algorithm (NSGA)

L'idea è che nella funzione monobiettivo posso costruire una ranked list degli individui, potendoli ordinare in base alla probabilità di sopravvivenza. Questa ranked list non è costruibile per i MOPs, questo perché due elementi non sono confrontabili se una soluzione è migliore in una funzione e un'altra soluzione in un'altra funzione. NSGA va quindi a costruire tanti fronti di Pareto. È come se costruisse dei layer:

- Al primo ci sono quelli che non si dominano l'uno con l'altro ma dominano quelli al secondo layer. Esse non sono dominate da nessuno.
- Al secondo layer non si dominano tra di loro ma dominano quelli del terzo. Ciò fino ad arrivare alla fine degli individui. Di volta in volta, per identificare nuovi layer, le soluzioni dei layer precedenti vengono rimosse. Costruito un ordinamento in tale modo, dovendo prendere parte della popolazione, parto dal primo layer fin quando non arrivo alla parte di popolazione necessaria

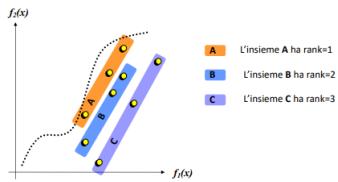


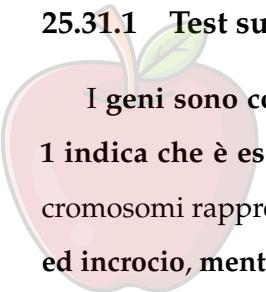
Figura 25.11: NSGA

25.30 Fast NSGA

Algoritmo utilizzato per i MOPs, in questa forma ciò che cambia è la fase di selezione. In tale fase si individuano i Fronti di Pareto, assegnando poi ai rank più alti una maggior probabilità di sopravvivenza. Il resto del GA funziona normalmente. Ad ogni fase, si andrà a calcolare il nuovo NSGA con relativi nuovi rank per gli individui.

25.31 GAs e testing di regressione

25.31.1 Test suite minimization & test case selection

I geni sono combinazioni di 0 ed 1, dove 0 indica che il test case non è eseguito ed 1 indica che è eseguito. Andando ad applicare crossover, dal momento che ognuno dei cromosomi rappresenta un numero di casi di test della test suite, prendo un punto di taglio ed incrocio, mentre con la mutation un test case passa da 0 ad 1, ovvero che se prima non lo prendevo ora lo prendo e viceversa

25.32 Test suite prioritization

Nella prioritization, mediante GAs, ogni cromosoma mi dice per ogni caso di test qual'è la sua posizione nella lista prioritizzata, dove il numero di posizioni è pari al numero di test case. Ogni posizione deve comparire una sola volta. Presa una popolazione casuale, devo prendere un ordinamento qualunque dei numeri che vanno da 1 ad n (supponendo n test cases) ed una permutazione dei valori che vanno da 1 ad n, ciò mi andrà ad indicare le posizioni nella ranked list.

Il crossover viene fatto andando ad applicare single point crossover, andando poi a verificare che nei nuovi cromosomi venga rispettata la condizione sulla permutazione, ovvero che un valore può occorrere una sola volta, altrimenti si possono usare operatori più sofisticati. La mutazione viene fatta scambiando due valori nella ranked list.

Rappresentazione - Test Suite Minimization and Test Case Selection

1	0	0	0	1	0	1	1	1
1	1	0	1	1	0	0	1	0

Rappresentazione - Test Case Prioritization

3	5	1	4	7	2	8	6	9
8	5	1	9	6	2	3	7	4

Figura 25.12: Test suite prioritization

25.33 Ipervolume

All'interno del problema della prioritizzazione, per passare da una MOP ad un problema monobiettivo, si è usato un indicatore, chiamato l'ipervolume. Con gli algoritmi greedy, l'area sotto la curva era in qualche modo quella che veniva massimizzata. La soluzione migliore, soprattutto nella prioritizzazione, era quella che andava a massimizzare l'area sotto la curva. L'ipervolume è proprio un qualcosa che va a calcolare questo, ovvero si basa la funzione obiettivo su quell'area. Facendo ciò si riesce a trasformare un problema multiobiettivo in un problema monobiettivo.

25.34 GAS: conclusioni

Normalmente, in pratica, applicando algoritmi genetici e abbiamo una popolazione, se vediamo cosa succede nel tempo possiamo osservare come la popolazione abbia una direzione, spostandosi da una parte dello spazio di ricerca ad un'altra. Potrei quindi stimare un'evoluzione futura sulla base della direzione. Se questa direzione mi porta verso un ottimo locale, ad un certo punto rimarrò intrappolato. Per quanto gli algoritmi genetici siano algoritmi di ricerca globale, in quanto sulla base dei parametri potrebbero avvantaggiare di più exploration e non exploitation, ciò potrebbe non essere sufficiente. Per scappare dalla direzione che sta prendendo l'evoluzione, se riuscissi a stimare tale direzione, potrei in qualche modo far deviare la popolazione, andando a seguire dei percorsi ortogonali. L'idea è quindi quella di far evolvere per al più un certo numero di volte la popolazione di partenza. L'idea è che una popolazione di soluzioni fornite dall'algoritmo genetico ad una generazione t, può essere vista come una matrice mxn.

25.35 Single value decomposition

Questa tecnica va a **decomporre la matrice in 3 matrici**. La prima mi va a proiettare i termini dallo spazio dei documenti allo spazio K di dimensione rxr, dove r è il rango della matrice mxn, e rappresenta il numero di valori singolari. T_0 sarà la matrice dei vettori singolari di sinistra mentre D_0 è la matrice dei vettori singolari di destra. Quello che posso fare è permutare tutti i valori singolari in modo tale da avere in posizione (1, 1) quello più in alto e man mano a scendere.

