

Ingegneria del Software

Bumma Giuseppe

Contents

1 Diagrammi UML	12
1.1 Premessa	12
1.2 Unified Modeling Language	13
1.3 Diagrammi di UML 2.5.1	13
1.4 UML 2.5.1 e Visio	14
1.5 Diagramma dei Package	14
1.5.1.1 Package	14
1.5.1.2 Diagramma dei Package	14
1.5.1.3 Dipendenze	15
1.5.1.4 Diagramma dei Package	15
1.5.1.5 Esempio	16
1.6 Interfaccia	16
1.6.1.1 Notazione	17
1.7 Diagramma delle Classi	17
1.7.1.1 Esempio	17
1.7.1.2 Classe	18
1.7.2 Attributi	18
1.7.2.1 Molteplicità	19
1.7.2.2 Visibilità	19
1.7.2.3 Attributi: Molteplicità	19
1.7.2.4 Operazioni	19
1.7.2.5 Operazioni e Attributi Statici	20
1.7.2.6 Associazioni	20
1.7.2.7 Associazioni Bidirezionali	21
1.7.2.8 Associazioni Ternarie	21
1.7.2.9 Associazioni: molteplicità	21
1.7.3 Classi di Associazione	22
1.7.4 Aggregazione e Composizione	22
1.7.5 Generalizzazione	23
1.7.6 Generalizzazione Multipla	24
1.7.6.1 Relazione tra classi: sintassi	25
1.7.7 Classi Astratte	25
1.7.8 Enumerazioni	25
1.8 Diagramma di sequenza	25
1.8.1 Esempio	26
1.8.2 Lifeline	26
1.8.2.1 Vincoli Temporali	27
1.8.3 Riferimento ad altri Diagrammi	28
1.8.3.1 Messaggio	29
1.8.3.1.1 Tipi di Messaggio	29
1.8.4 Combined Fragment	31
1.8.4.1 Tipi	31
1.9 Diagrammi di Stato	33

1.9.1.1 Esempio	33
1.9.1.2 Concetti	34
1.9.1.3 Esempio	34
1.9.1.4 Stato	35
1.9.1.5 Esempio	36
1.9.1.6 Esempio	36
1.9.1.7 Esempio	37
1.9.1.8 Pseudostato	37
1.9.1.9 Tipi di Pseudostati	37
1.9.1.10 Transizione	39
1.9.1.11 Tipi di Eventi	40
1.9.1.12 Azione	40
1.9.1.13 Diagramma di Stato	41
1.10 Diagramma delle attività	42
1.10.1.1 Notazione	43
1.10.2 Object Flow	44
1.10.3 Segnali ed Eventi	44
1.10.4 Exception Handler	44
1.10.5 InterruptibleActivityRegion	46
2 Progetto	46
2.1 Analisi dei requisiti	46
2.1.1 Requisiti	46
2.1.1.1 Requisiti di Sistema	47
2.1.1.2 Requisiti Funzionali	47
2.1.1.3 Requisiti Non Funzionali	47
2.1.1.4 Requisiti di Dominio	48
2.1.2 Analisi	48
2.1.2.1 Raccolta dei Requisiti	48
2.1.2.2 Validazione dei Requisiti	49
2.1.2.3 Documento dei Requisiti	49
2.1.2.4 Cambiamento dei Requisiti	50
2.1.2.5 Analisi del Dominio	50
2.1.2.6 Analisi dei Requisiti	51
2.1.2.7 Vocabolario	51
2.1.2.8 Analisi e gestione dei rischi	51
2.1.3 Sicurezza e Privacy	52
2.1.3.1 Nuova normativa	52
2.1.3.1.1 General Data Protection Regulation	52
2.1.3.1.2 Principi	53
2.1.3.1.3 Articolo 25	53
2.1.3.1.4 Articolo 32:	53
2.1.3.1.5 Trasferimento Dati a Paesi Terzi	54
2.1.3.2 Concetti di Base	54
2.1.3.2.1 Sicurezza Informatica	54
2.1.3.2.2 Violazioni	55
2.1.3.2.3 Fattori Influenti	55
La Catena degli Anelli	55
2.1.3.2.4 Metodi di Protezione	55

2.1.3.2.5 Protezione Fisica	56
2.1.3.2.6 Autenticazione Forte	56
2.1.3.3 Crittografia	57
2.1.3.3.1 Crittografia Simmetrica	57
2.1.3.3.2 Crittografia Asimmetrica	57
2.1.3.3.3 Confronto tra i due tipo di crittografia	58
2.1.3.3.4 Biometria	59
2.1.3.3.5 Sicurezza delle Password	59
2.1.3.4 Analizzare e Progettare la Sicurezza	59
2.1.3.4.1 Sistema Sicuro	59
2.1.3.4.2 Sistemi Critici	60
2.1.3.5 Introduzione alla Security Engineering	61
2.1.3.5.1 Security Engineering	61
2.1.3.5.2 Applicazione e Infrastruttura	62
2.1.3.5.3 Gestione della Sicurezza	62
2.1.3.6 Glossario e Minacce	62
2.1.3.6.1 Glossario della Sicurezza	62
2.1.3.6.2 Tipi di Minacce	63
2.1.3.6.3 Tipi di Controllo	63
Esempio	63
2.1.3.7 Analisi del Rischio	64
2.1.3.8 Identificazione del bene	65
2.1.3.8.1 Analisi del Sistema Informatico	65
2.1.3.8.2 Analisi delle Risorse Fisiche	65
2.1.3.8.3 Analisi delle Risorse Logiche	66
2.1.3.8.4 Analisi delle Dipendenze tra Risorse	66
2.1.3.9 Identificazione delle minacce	66
2.1.3.9.1 Attacchi Intenzionali	66
2.1.3.9.2 Attacchi a Livello Logico	66
2.1.3.9.3 Eventi Accidentali	67
2.1.3.9.4 Valutazione dell'Esposizione	67
2.1.3.9.5 Valutazione delle Probabilità: Attacchi Intenzionali	67
2.1.3.9.6 Individuazione del Controllo	67
2.1.3.9.7 Valutazione del Rapporto Costo/Efficacia	67
2.1.3.9.8 Costi Nascosti	68
2.1.3.9.9 Controlli di Carattere Organizzativo	68
2.1.3.9.10 Controlli di Carattere Tecnico	68
2.2 Analisi del problema	78
2.2.1 Introduzione	78
2.2.2 Passi dell'Analisi del Problema	79
2.2.2.1 Architettura Logica	80
2.2.2.2 Piano di Lavoro	80
2.2.2.3 Piano del Collaudo	80
2.2.3 Analisi Documento dei Requisiti	82
2.2.3.1 Analisi delle Funzionalità	82
2.2.3.2 Esempio	83
2.2.3.3 Analisi dei Vincoli	84
2.2.3.4 Esempio	85

2.2.3.5 Analisi delle Interazioni	85
2.2.3.6 Esempio	86
2.2.3.7 Esempio	86
2.2.4 Analisi Ruoli e Responsabilità	87
2.2.4.1 Analisi dei Ruoli e Responsabilità	87
2.2.4.2 Esempio	87
2.2.4.3 Esempio	88
2.2.5 Scomposizione del Problema	88
2.2.6 Creazione Modello del Dominio	89
2.2.6.1 Individuazione delle Classi	90
2.2.6.2 Individuazione delle Classi: Villaggio Turistico	91
2.2.6.3 Individuazione delle Relazioni	92
2.2.6.4 Individuazione dell'Ereditarietà	93
2.2.6.5 Ereditarietà: Villaggio Turistico	93
2.2.6.6 Individuazione delle Associazioni	93
2.2.7 Individuazione delle Associazioni	94
2.2.7.1 1 ° Esempio di Associazione	94
2.2.7.2 2 ° Esempio di Associazione	95
2.2.7.3 Associazioni: Villaggio Turistico	95
2.2.7.4 Individuazione Collaborazioni	96
2.2.7.5 Individuazione degli Attributi	97
2.2.7.6 Individuazione degli Attributi: Villaggio Turistico	99
2.2.7.7 Individuazione delle Operazioni	100
2.2.7.8 Esempio	101
2.2.8 Architettura Logica: Struttura	103
2.2.8.1 BCE	103
2.2.8.2 Layer	104
2.2.8.3 Struttura: Package	104
2.2.8.4 Esempio	106
2.2.8.5 Struttura: Classi	106
2.2.8.6 Esempio	106
2.2.9 Architettura Logica: Interazione	107
2.2.9.1 Esempio	107
2.2.10 Architettura Logica: Comportamento	108
2.2.10.1 Esempio	108
2.2.11 Definizione del Piano di Lavoro	109
2.2.11.1 Esempio	109
2.2.12 Definizione del Piano del Collaudo	110
2.2.12.1 Definizione Piano del Collaudo	110
2.2.12.2 JUnit	111
2.2.12.3 NUnit	111
2.3 Progettazione	111
2.3.1 Introduzione	111
2.3.2 Progettazione Architetturale	112
2.3.2.1 Requisiti Non Funzionali	113
2.3.2.2 Esempio	114
2.3.2.3 Scelta Architettura	114
2.3.2.3.1 Blackboard	115

2.3.2.3.2 MVC	115
2.3.2.3.3 Layer	115
2.3.2.3.4 Client/Server	116
2.3.2.3.5 Broker	116
2.3.2.3.6 Pipe & Filters	117
2.3.2.3.7 Conclusioni	117
2.3.2.3.8 Esempio: Villaggio Turistico	117
2.3.2.4 Scelte Tecnologiche	118
2.3.3 Progettazione di dettaglio	118
2.3.3.1 Architettura: Struttura	119
2.3.3.2 Struttura: Esempio	120
2.3.3.3 Architettura: Interazione	123
2.3.3.3.1 Interazione: Esempio	123
2.3.4 Architettura: Comportamento	124
2.3.5 Progettazione della persistenza	124
2.3.5.1 Quando usare un DBMS	125
2.3.5.2 Esempio: DB Villaggio Turistico	126
2.3.5.3 Esempio: log Villaggio Turistico	126
2.3.6 Progettazione del collaudo	126
2.3.7 Progettazione per il deployment	127
2.3.7.1 Deployment: Esempio	127
2.3.8 Design Pattern	127
2.3.8.1 L'importanza dei nomi dei Pattern	128
2.3.8.2 Classificazione dei Design Pattern	128
2.3.8.3 Pattern SINGLETON	129
2.3.8.4 Pattern OBSERVER	130
2.3.8.4.1 Esempio Boss-Worker	131
Class-based callback relationship	131
Interface-based callback relationship	131
Pattern Observer (lista di notifiche)	132
2.3.8.5 Pattern Model / View / Controller (MVC)	132
2.3.8.6 Pattern FLYWEIGHT	133
2.3.8.6.1 Esempio	134
2.3.8.7 Pattern STRATEGY	135
2.3.8.7.1 Esempio	136
2.3.8.8 Pattern ADAPTER	136
2.3.8.8.1 Esempio	137
2.3.8.9 Pattern DECORATOR	137
2.3.8.10 Ereditarietà Dinamica	139
2.3.8.11 Pattern STATE	141
2.3.8.12 Pattern COMPOSITE	141
2.3.8.13 Pattern VISITOR	145
2.3.8.14 Anti Pattern	149
2.3.8.15 Pattern ABSTRACT FACTORY	149
2.4 Diagramma dei componenti e di Deployment	151
2.4.1 Diagramma dei Componenti	151
2.4.1.1 Struttura Composita	151
2.4.1.2 Package vs. Struttura Composita	152

2.4.1.3 Componenti: White-box	153
2.4.1.4 Connettori Multiple Wiring	154
2.4.1.5 Componenti e sottosistemi	155
2.4.1.6 Il diagramma	155
2.4.1.7 Esempio Villaggio Turistico	155
2.4.2 Diagramma di Deployment	156
2.4.2.1 Esempio Villaggio Turistico	157
2.4.2.2 Esempio Villaggio Turistico	159
2.5 Progettazione Concettuale (E/R)	159
2.5.1 Raccolta dei requisiti	160
2.5.2 Interagire con gli utenti	160
2.5.3 Requisiti: documentazione descrittiva	160
2.5.3.1 Esempio: BD bibliografica (1)	161
2.5.3.2 Esempio: BD bibliografica (2)	161
2.5.3.3 Un altro esempio più articolato	161
2.5.3.4 Glossario dei termini, omonimi e sinonimi	161
2.5.3.4.1 Esempio	162
2.5.3.5 Ristrutturazione dei requisiti	162
2.5.3.5.1 Esempio: frasi relative ai partecipanti	162
2.5.3.6 Dai concetti allo schema E/R	162
2.5.3.7 Strategie di progettazione	163
2.5.3.7.1 Pro e controllo delle strategie	163
2.5.3.7.2 Un approccio “misto”	163
2.5.3.7.3 Società di formazione: schema scheletro	163
Raffinamento di Partecipanti	164
Raffinamento di Corsi	164
Raffinamento di Docenti	165
Integrazione: schema di riferimento	165
Integrazione: Partecipanti e Corsi	165
Integrazione: Docenti e Corsi	166
2.5.3.8 Qualità di uno schema concettuale	166
2.5.3.9 Metodologia basata sulla strategia mista	166
2.5.3.10 Riassumendo	167
3 Framework .NET	167
3.1 Introduzione	167
3.1.1 Tecnologia COM - Component Object Model	167
3.1.2 Cos’è il Framework .NET	168
3.1.3 Standard ECMA-335	169
3.1.4 Codice interpretato	170
3.1.5 Codice nativo	170
3.1.6 Codice IL	170
3.1.7 Assembly	171
3.1.8 Metadati	172
3.1.9 Chi usa i metadati?	172
3.1.10 Esempio Assembly	173
3.1.11 Dove trovare gli Assembly	173
3.1.12 Deployment semplificato	173
3.1.13 Common Language Runtime	173

3.1.14 Garbage Collector	174
3.1.15 Gestione delle eccezioni	175
3.1.16 Common Type System	175
3.1.17 Common Language Specification	176
3.1.17.1 Tipi nativi	176
3.1.18 Common Type System	176
3.1.18.1 Tipi valore	177
3.1.18.2 Tipi valore vs tipi riferimento	177
3.1.18.3 Common Type System	177
3.1.18.4 Tipi valore e tipi riferimento	179
3.1.19 Boxing / Unboxing	180
3.1.20 Bibliografia	180
3.2 Garbage Collection	181
3.2.1 Utilizzo di un oggetto	181
3.2.2 Ciclo di vita di un oggetto	181
3.2.3 Allocazione della memoria	181
3.2.4 Inizializzazione della memoria	182
3.2.5 Definite Assignment	182
3.2.6 Clean up dello stato	183
3.2.7 Liberazione della memoria	183
3.2.8 Cos'è il Garbage Collection	183
3.2.9 GC: Reference counting	184
3.2.10 GC: Tracing	184
3.2.11 Allocazione della memoria	185
3.2.12 Garbage Collector	186
3.2.12.1 Fase 1: Mark	186
3.2.12.2 Fase 2: Compact	187
3.2.13 Finalization	187
3.2.14 Rilascio deterministico senza gestione delle eccezioni	189
3.2.15 Rilascio deterministico con gestione delle eccezioni	189
3.2.16 Il pattern Dispose	189
3.2.17 Rilascio deterministico con using	189
3.2.18 Il pattern Dispose (altro esempio di utilizzo)	190
3.3 Tipi in .NET	190
3.3.1 Modificatori di visibilità	191
3.3.2 Regole	191
3.3.3 Costanti	192
3.3.4 Field	192
3.3.5 Regole	193
3.3.6 Modificatori di metodi	193
3.3.7 Modificatore virtual	193
3.3.8 Modificatore abstract	194
3.3.9 Modificatore override	194
3.3.10 Metodi	194
3.3.10.1 Passaggio degli argomenti	194
3.3.10.2 Passaggio degli argomenti In	195
3.3.10.3 Passaggio degli argomenti In/Out	195
3.3.10.4 Passaggio degli argomenti	195

3.3.10.5 Passaggio degli argomenti Out	196
3.3.10.6 Regole	197
3.3.10.7 Esempi	197
3.3.10.8 Numero variabile di argomenti	197
3.3.11 Costruttori di istanza	198
3.3.12 Costruttori di tipo	199
3.3.13 Regole	199
3.3.14 Costruttori ed eccezioni	200
3.3.15 Interfacce	200
3.3.15.1 Implementazione di un’interfaccia	200
3.3.15.2 Implementazione di un’interfaccia: classe astratta	201
3.3.15.3 Implementazione esplicita di un’interfaccia	201
3.3.16 Interfaccia vs. Classe astratta	203
3.4 Classi e Interfacce Base	204
3.4.1 Framework .NET: Overview	204
3.4.2 System.Object	204
3.4.3 Object.Equals	205
3.4.4 Object.Equals	206
3.4.5 System.ValueType	207
3.4.6 System.Boolean	208
3.4.7 System.Int32	208
3.4.8 System.IComparable	208
3.4.9 System.IComparable	209
3.4.9.1 System.Collections.IComparer	209
3.4.10 System.IConvertible	209
3.4.11 System.Convert	210
3.4.12 System.Convert	211
3.4.13 Conversione di tipo	211
3.4.13.1 Conversione di tipo definite dall’utente	213
3.4.14 Conversioni a string	213
3.4.15 Conversioni da string	213
3.4.16 Conversioni a/da string	214
3.4.17 System.Double	214
3.4.18 System.Enum	215
3.4.19 System.DateTime	215
3.4.20 System.String	216
3.4.21 System.ICloneable	216
3.4.21.1 System.Collections.IEnumerable	217
3.4.21.2 System.Collections.IEnumerator	217
3.4.22 System.Array	218
3.5 Delegati ed eventi	219
3.5.1 Delegati	219
3.5.1.1 C/C++: Puntatori a funzioni	220
3.5.1.2 C/C++: Array di puntatori a funzioni	220
3.5.1.3 C/C++: Elaborazione cooperativa	220
3.5.2 Delegati	220
3.5.2.1 Delegati: Multicasting	221
3.5.2.2 Esempio Boss-Worker	223

3.5.2.3 Una Relazione di Chiamata Basata su Delegati	223
3.5.3 Eventi	225
3.5.3.1 Dichiarazione di un Evento - Convenzione .NET	226
3.5.3.2 Invocazione di un Evento	226
3.5.3.2.1 Agganciarsi a un Evento	227
3.5.3.2.2 Sganciarsi da un Evento	227
4 Principi di Design	228
4.1 Qualità della progettazione	228
4.2 Cosa rende un design “cattivo”?	228
4.2.1 Rigidità del Software	228
4.2.2 Fragilità del Software	228
4.2.3 Immobilità del Software	228
4.2.4 Viscosità del Software	229
4.2.5 Perché esistono risultati di progettazione scadenti?	229
4.3 Modifiche ai Requisiti	229
4.4 Gestione delle Dipendenze	229
4.5 Principi di Design	230
4.5.1 Premessa	230
4.5.1.1 Il principio zero	230
4.5.1.2 Semplicità e semplicismo	230
4.5.1.3 Divide et impera	230
4.5.1.4 Rendere privati tutti i dati degli oggetti	230
4.5.2 The Single Responsibility Principle	231
4.5.2.1 Esempio	231
4.5.2.2 Esempio - Refactoring	232
4.5.3 The Dependency Inversion Principle	232
4.5.3.1 Dipendenze transitive	233
4.5.3.2 Dipendenze cicliche	234
4.5.3.3 Stabilità delle astrazioni	234
4.5.4 The Interface Segregation Principle	235
4.5.4.1 Fat Interface	235
4.5.5 The Interface Segregation Principle	235
4.5.5.1 Segregated Interfaces	235
4.5.6 The Open/Closed Principle	236
4.5.6.1 Esempio 1	237
4.5.6.2 Esempio 2	239
4.5.6.3 Esempio 3	240
4.5.6.4 Esempio 4	241
4.5.6.5 Conlcusioni	241
4.5.7 The Liskov Substitution Principle	242
4.5.7.1 Example	242
4.6 Design by Contract	243
4.7 Il Quadrato è un Rettangolo?	243
4.8 Il Quadrato è un Rettangolo?	248
4.9 Principi di Architettura dei Package	248
4.9.1 Release/Reuse Equivalency Principle	248
4.9.2 Common Closure Principle	249
4.9.3 Common Reuse Principle	249

4.9.4 Discussione	249
4.10 Relazioni tra i Package	250
4.10.1 Acyclic Dependencies Principle	250
4.10.1.1 Esempio: Grafo dei Package Aciclico	250
4.10.1.2 Esempio: Grafo dei Package Ciclico	250
4.10.1.3 Discussione	251
4.10.1.4 Rompere il Ciclo Introducendo un'Interfaccia	251
4.10.2 Stable Dependencies Principle	251
4.10.2.1 Esempio	252
4.10.3 Stable Abstractions Principle	252
4.10.3.1 Esempio	252
4.10.4 Stable Dependencies/Abstractions Principles	252
4.10.4.1 Discussione	252
5 Sicurezza	253
5.1 Progettazione per la Sicurezza	253
5.2 Progettazione Architetturale	253
5.2.1 Esempio	254
5.2.2 Esempio	254
5.2.3 Esempio	255
5.3 Linee Guida di Progettazione	255
5.3.1 Basare la Sicurezza su Policy	256
5.3.1.1 Progettazione delle Politiche	256
5.3.2 Evitare Punto Singolo di Fallimento	257
5.3.3 Fallire in Modo Certo	257
5.3.4 Bilanciare Sicurezza e Usabilità	257
5.3.5 Essere Consapevoli dell'Ingegneria Sociale	258
5.3.6 Usare Ridondanza e Diversità	258
5.3.7 Validare tutti gli Input	258
5.3.8 Dividere in Compartimenti i Beni	258
5.3.9 Progettazione per il Deployment	259
5.3.10 Progettazione per il Ripristino	259
5.4 Progettazione per il Deployment	259
5.4.1 Deployment del Software	259
5.4.2 Linee Guida	260
5.4.2.1 Supporto per le Configurazioni	260
5.4.2.2 Minimizzare i Privilegi di Default	261
5.4.2.3 Localizzare le Impostazioni di Configurazione	261
5.4.2.4 Rimediare a Vulnerabilità	261
5.5 Testare la Sicurezza	261
5.5.1 Black Box Testing	262
5.5.2 White Box Testing	262
5.6 Capacità di Sopravvivenza del Sistema	263
5.6.1 Esempio	263
5.6.2 Analisi della Sopravvivenza	263
5.6.3 Strategie	263
5.6.4 Fasi Analisi di Sopravvivenza	264
5.6.5 Principali Attività	264
5.6.6 Esempio	264

5.6.7 Esempio	265
5.6.8 Esempio	265
5.6.9 Esempio	265
5.6.10 Esempio	266
5.6.11 Capacità di Sopravvivenza	266
5.7 Conclusioni	266
6 Dalla Progettazione all'Implementazione	266
6.1 Introduzione	266
6.2 Progettazione di Dettaglio	267
6.2.1 Navigabilità di un'Associazione	267
6.2.2 Implementazione delle Associazioni	268
6.2.2.1 Associazioni con molteplicità 0..1 o 1..1	268
6.2.2.2 Associazioni con molteplicità 0..* o 1..*	268
6.2.3 Classi Contenitore	270
6.2.3.1 Contenimento per Riferimento	270
6.2.3.2 Contenimento per Valore	271
6.2.3.3 Contenimento di Oggetti Omogenei	271
6.2.3.4 Contenimento di Oggetti Eterogenei	271
6.3 Implementazione delle Associazioni	272
6.3.1 Identificazione degli Oggetti	272
6.3.1.1 Un Esempio Reale	273
6.3.1.2 Modifiche per Utilizzare il Livello di Ereditarietà Supportato	274
6.3.1.2.1 1 ^a possibilità (composizione e delega)	274
6.3.1.2.2 2 ^a possibilità (interfaccia)	275
6.4 Miglioramento delle Prestazioni	276

1 Diagrammi UML

1.1 Premessa

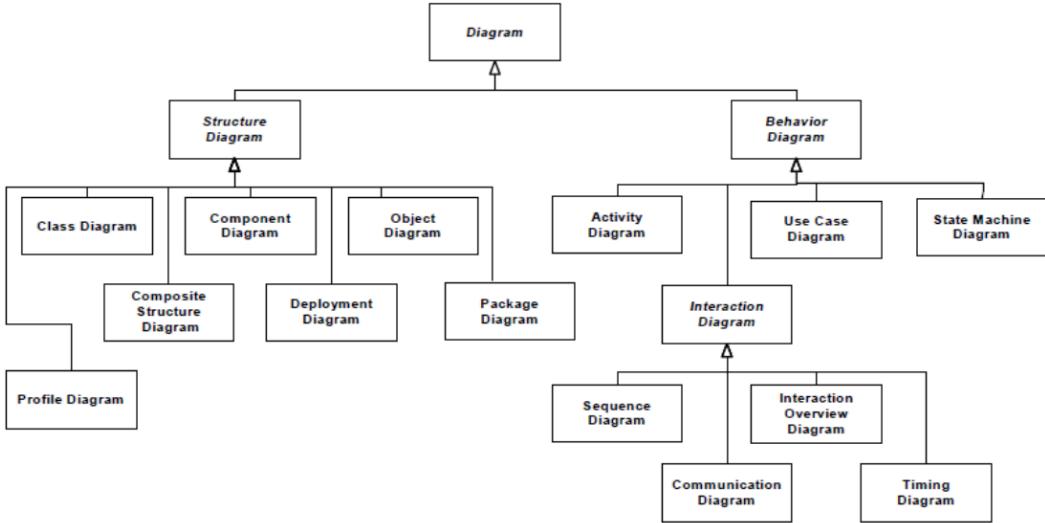
- In questo blocco vedremo i principali Diagrammi UML per la fasi di Analisi del Problema e del Progetto
 - ▶ per ora non vedremo i diagrammi di deployment e dei componenti
- In particolare:
 - ▶ Diagramma dei Package e Diagramma delle Classi → parte “**statica**” dell’Architettura Logica e dell’Architettura del Sistema
 - definiscono le entità di sistema senza analizzare come possano interagire tra loro
 - ▶ Diagramma di Sequenza → parte “**interazione**” dell’Architettura Logica e dell’Architettura del Sistema
 - definiscono come le entità, definite nella parte precedente, interagiscono tra loro
 - ▶ Diagramma di Stato e Diagramma delle Attività → parte “**comportamentale**” dell’Architettura Logica e dell’Architettura del Sistema
 - definiscono il comportamento delle entità del sistema
- Per ogni Diagramma vedremo solo i concetti fondamentali
- In caso di dubbi fare riferimento alla specifica UML 2.5.1

1.2 Unified Modeling Language

- È un **linguaggio** che serve per visualizzare, specificare, costruire, documentare un sistema e gli elaborati prodotti durante il suo sviluppo
- Ha una semantica e una notazione standard, basate su un metamodello integrato, che definisce i costrutti forniti dal linguaggio
- La notazione (e la semantica) è estensibile e personalizzabile
- È utilizzabile per la modellazione durante tutto il ciclo di vita del software (dai requisiti al testing) e per piattaforme e domini diversi
- Combina approcci di:
 - modellazione dati (Entity/Relationship)
 - business Modeling (workflow)
 - modellazione a oggetti
 - modellazione di componenti
- Prevede una serie di diagrammi standard, che mostrano differenti viste architettoniche del modello di un sistema
- UML è un linguaggio e non un processo di sviluppo
- UML propone un ricco insieme di elementi a livello utente; tuttavia è alquanto informale sul modo di utilizzare al meglio i vari elementi
 - ciò implica che per comprendere un diagramma un lettore deve conoscere il contesto in cui esso è collocato

1.3 Diagrammi di UML 2.5.1

- Diagrammi di struttura:
 - diagramma delle classi (class)
 - diagramma delle strutture composite (composite structure)
 - diagramma dei componenti (component)
 - diagramma di deployment (deployment)
 - diagramma dei package (package)
 - diagramma dei profili (profile)
- Diagrammi di comportamento:
 - diagramma dei casi d'uso (use case)
 - diagramma di stato (state machine)
 - diagramma delle attività (activity)
 - diagrammi di interazione:
 - diagramma di comunicazione (communication)
 - diagramma dei tempi (timing)
 - diagramma di sintesi delle interazioni (interaction overview)
 - diagramma di sequenza (sequence)



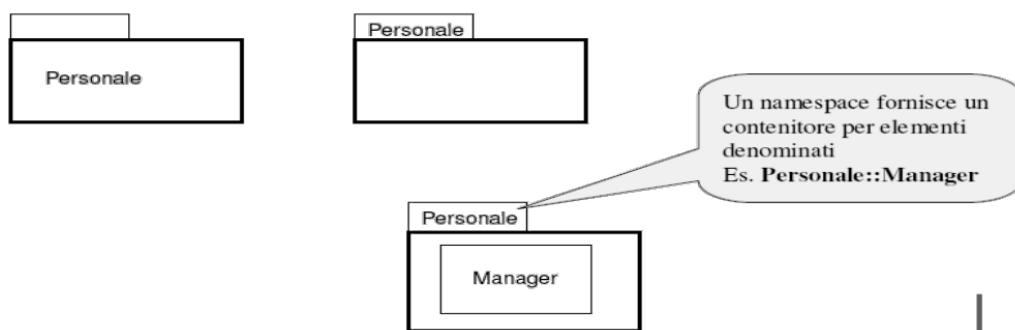
1.4 UML 2.5.1 e Visio

- Visio 2016 gestisce nativamente solo alcuni dei Diagrammi di UML 2.5.1
 - classi
 - attività
 - sequenza
 - stato
 - casi d'uso
- È possibile però scaricare uno stencil per poter gestire anche gli altri diagrammi:
<http://softwarestencils.com/uml/index.html>

1.5 Diagramma dei Package

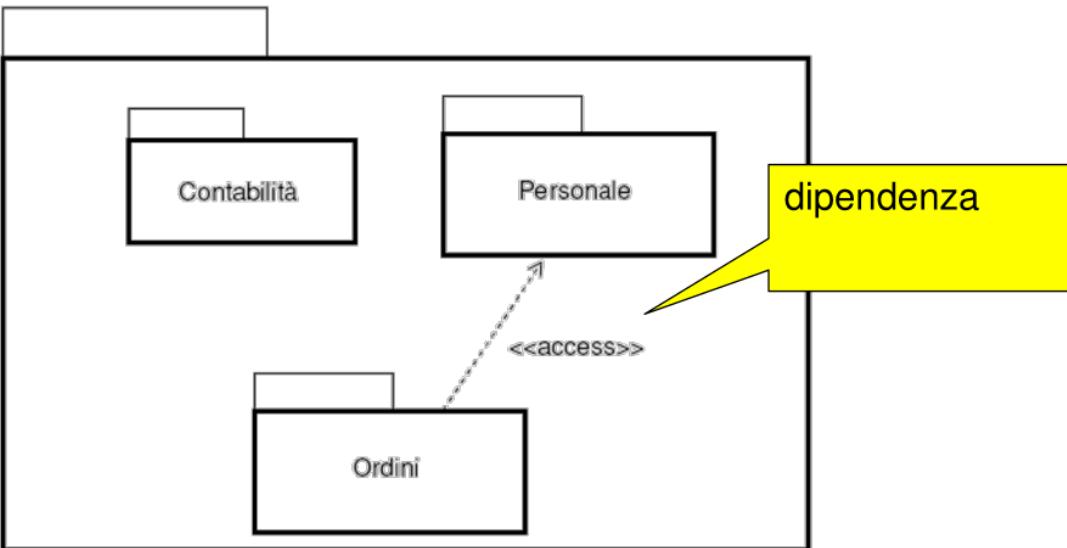
1.5.1.1 Package

- Un package, in programmazione, è un contenitore di classi
- Un package, in generale, è utilizzato per raggruppare elementi e fornire loro un namespace
- Un package può essere innestato in altri package
- NAMESPACE: è una porzione del modello nella quale possono essere definiti e usati dei nomi
- In un namespace ogni nome ha un significato univoco



1.5.1.2 Diagramma dei Package

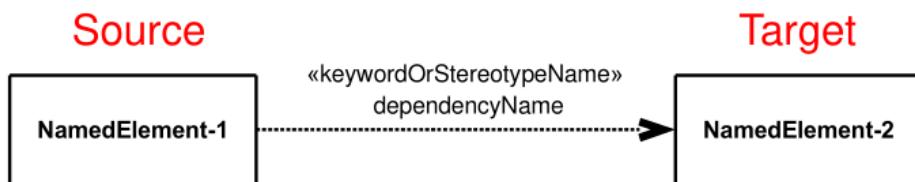
- Un diagramma dei package è un diagramma che illustra come gli elementi di modellazione sono organizzati in package e le relazioni (dipendenze) tra i package



- Un diagramma può rappresentare package con diversi livelli di astrazione

1.5.1.3 Dipendenze

- UML permette di rappresentare relazioni che NON sussistono fra istanze nel dominio rappresentato, ma sussistono fra gli elementi del modello UML stesso o fra le astrazioni che tali elementi rappresentano
- Dipendenza: è rappresentata da una linea tratteggiata orientata che va da un elemento dipendente (Source) ad uno indipendente(Target)



- Una dipendenza indica che cambiamenti dell'elemento **indipendente** influenzano l'elemento **dipendente**
 - modificano il significato dell'elemento dipendente
 - causano la necessità di modificare anche l'elemento dipendente perché i significati sono dipendenti
- UML mette a disposizione nove diversi tipi di dipendenza, ma per i nostri fini consideriamo quasi sempre <>

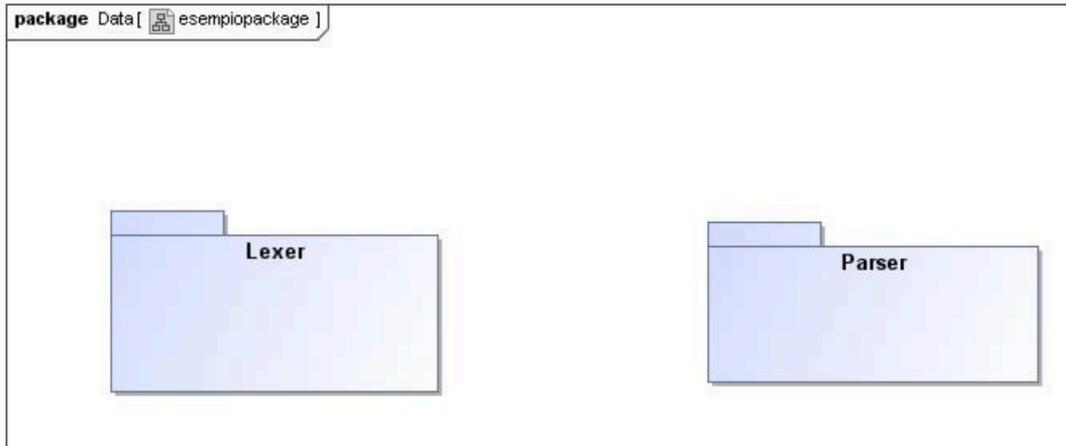


1.5.1.4 Diagramma dei Package

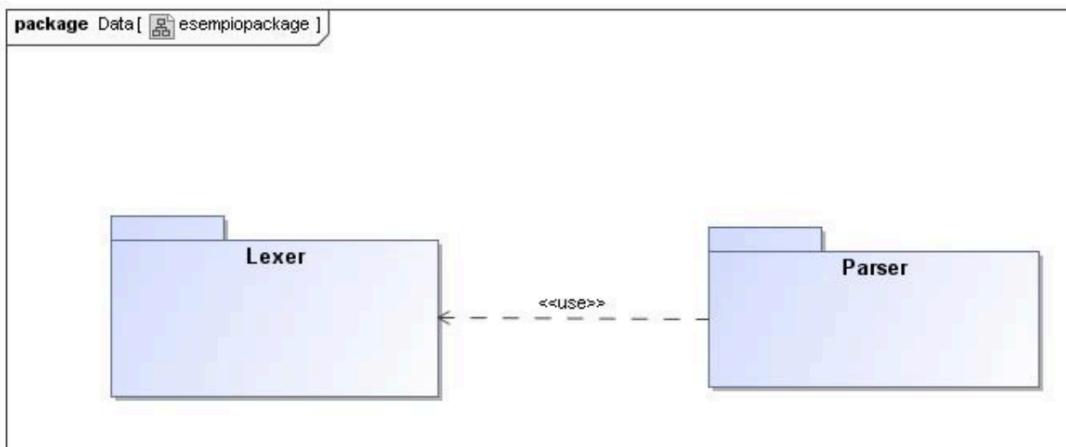
- Tutti i diagrammi che utilizzeremo nella fase di analisi saranno trasformati per venir utilizzati nella fase di progettazione (in analisi definisco il “cosa” e in progettazione definisco il “come”)
- Quando si usa il diagramma dei package per definire la parte strutturale dell'architettura logica ricordare sempre che si stanno esprimendo **dipendenze logiche** che sussistono tra le entità del problema
- Non è detto che tali dipendenze rimangano tali anche nella fase di progettazione

- Esempio: in sistema per l'interpretazione di una grammatica si hanno due parti fondamentali
 - ▶ Lexer → strumento che legge e spezza una sequenza di caratteri in sotto-sequenze dette "token"
 - ▶ Parser prende in ingresso i token generati, li analizza e li elabora in base ai costrutti specificati nella grammatica stessa

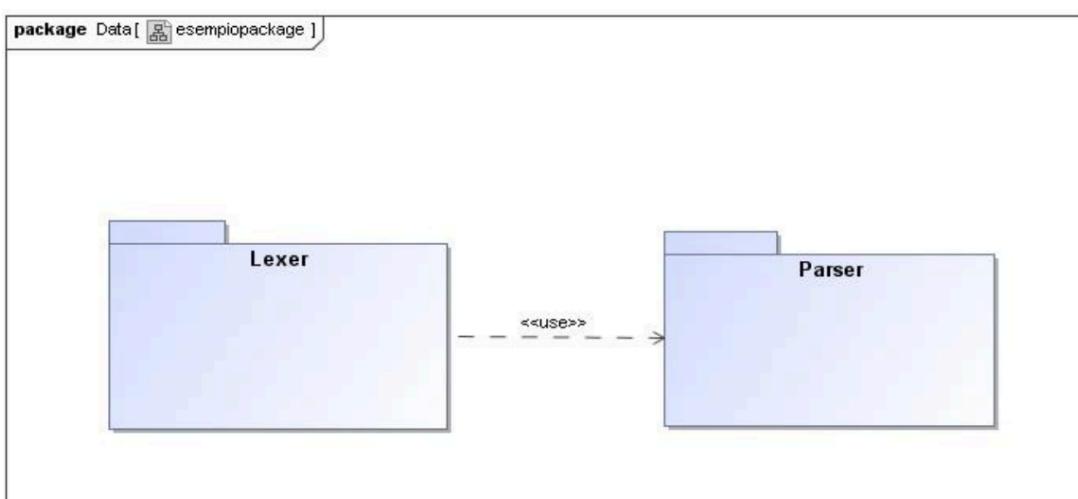
1.5.1.5 Esempio



È ovvio che è il Parser a usare il Lexer, quindi il diagramma prodotto nella fase di analisi sarà così



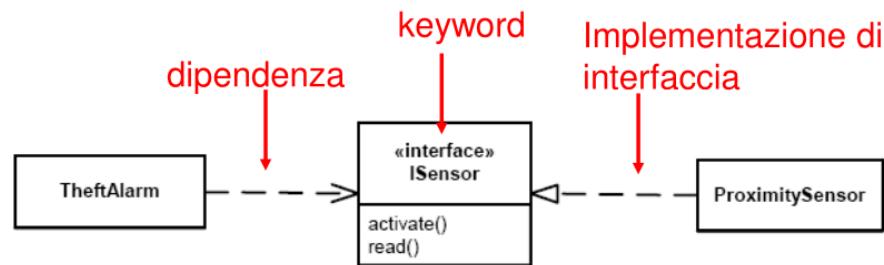
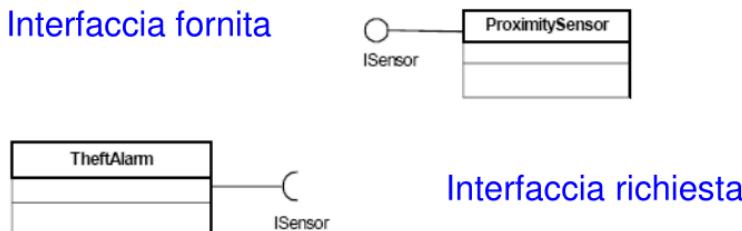
Ma nel diagramma del progetto è il Lexer a fare il Parser



1.6 Interfaccia

- Le interfacce forniscono un modo per partizionare e caratterizzare gruppi di proprietà
- Un’interfaccia non deve specificare come possa essere implementata, ma semplicemente quello che è necessario per poterla realizzare
- Le entità che realizzano l’interfaccia dovranno fornire una “**vista pubblica**”(attributi, operazioni, comportamento osservabile all’esterno) conforme all’interfaccia stessa
- Se un’interfaccia dichiara un attributo, non significa necessariamente che l’elemento che realizza l’interfaccia debba avere quell’attributo nella sua implementazione, ma solamente che esso apparirà così a un osservatore esterno

1.6.1.1 Notazione

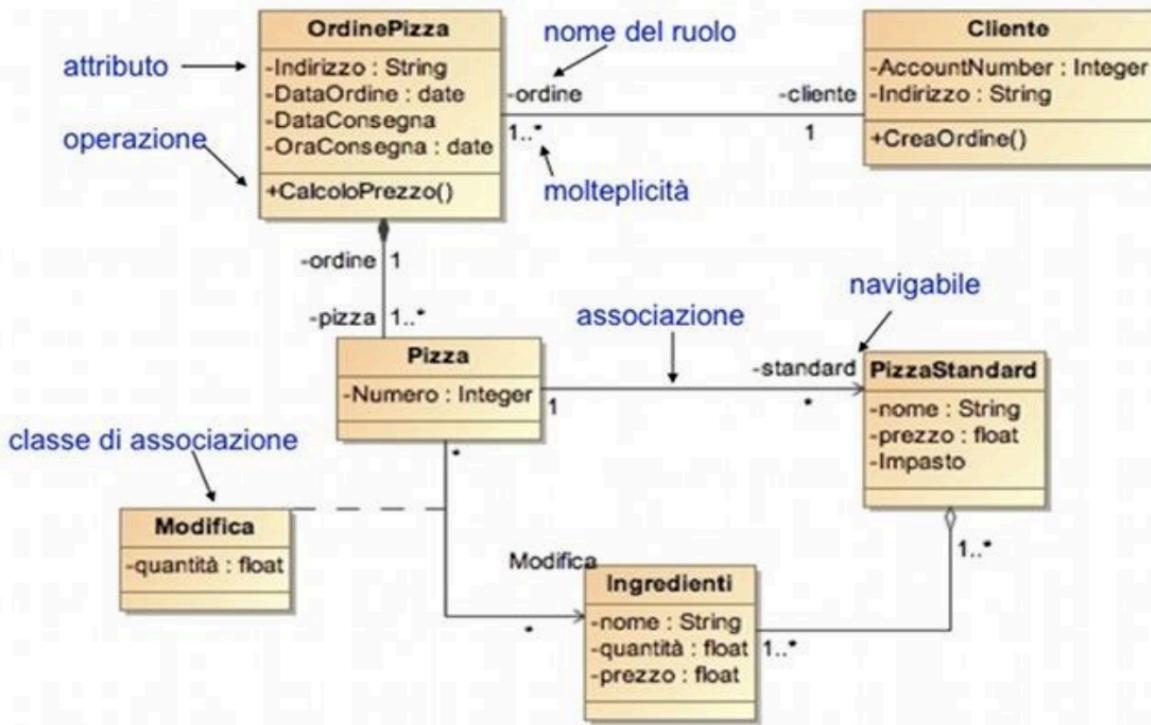


Il diagramma sopra e il diagramma sotto hanno lo stesso significato

1.7 Diagramma delle Classi

- Un diagramma delle classi descrive il tipo degli oggetti facenti parte di un sistema e le varie tipologie di relazioni statiche tra di essi
- I diagrammi delle classi mostrano anche le proprietà e le operazioni di una classe e i vincoli che si applicano alla classe e alle relazioni tra classi
- Le proprietà rappresentano le caratteristiche strutturali di una classe:
 - sono un unico concetto, rappresentato però con due notazioni molto diverse: attributi e associazioni
 - benché il loro aspetto grafico sia molto differente, concettualmente sono la stessa cosa

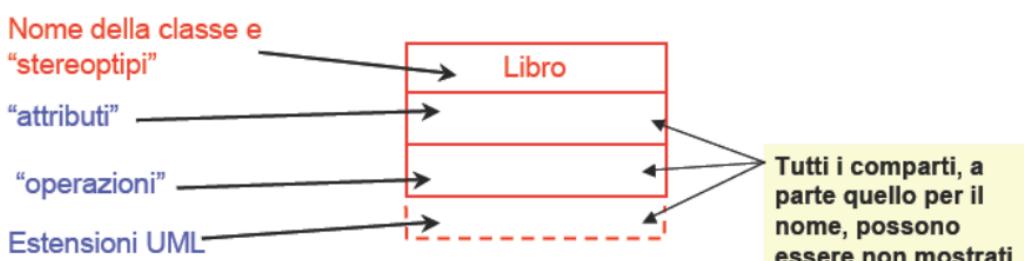
1.7.1.1 Esempio



- La molteplicità da OrdinePizza a Cliente è **uno a molti**: un cliente può avere da 1 a n ordini
- La molteplicità da Cliente a OrdinePizza è **univoca**: un ordine è relativo a uno e un solo cliente
- Associazione di **composizione** tra Pizza e OrdinePizza
- Associazione **navigabile** da Pizza a PizzaStandard: da Pizza ci interessa capire qual è la PizzaStandard, ma da PizzaStandard non ci interessa andare a Pizza, cioè ci interessa sapere ogni Pizza da quale PizzaStandard deriva
- Associazione di **aggregazione** tra PizzaStandard e Ingredienti
- Ogni Pizza può avere delle modifiche di ingredienti
- **Classe di associazione** Modifica, che aggiunge attributi e/o comportamenti a quell'associazione
 - ▶ Per ogni Pizza, ogni Ingrediente è presente in una certa quantità, che può essere modificata

1.7.1.2 Classe

- Una **classe** modella un insieme di entità (le istanze della classe) aventi tutti lo stesso tipo di caratteristiche (attributi, associazioni, operazioni...)
- Ogni classe è descritta da:
 - ▶ un nome
 - ▶ un insieme di caratteristiche (feature): attributi, operazioni, ...



1.7.2 Attributi

- La notazione degli attributi descrive una proprietà con una riga di testo all'interno del box della classe

- La forma completa è:
 - visibilità nome:tipo molteplicità==default {stringa di proprietà}
- Un esempio di attributo è:
 - stringa: String[10] == "Pippo" {readOnly}
- L'unico elemento necessario è il nome
 - Visibilità: attributo pubblico (+), privato (-) o protected(==)
 - Nome: corrisponde al nome dell'attributo
 - Tipo: vincolo sugli oggetti che possono rappresentare l'attributo
 - Default: valore dell'attributo in un oggetto appena creato
 - Stringa di proprietà: caratteristiche aggiuntive (readOnly)
 - Molteplicità: ...

1.7.2.1 Molteplicità

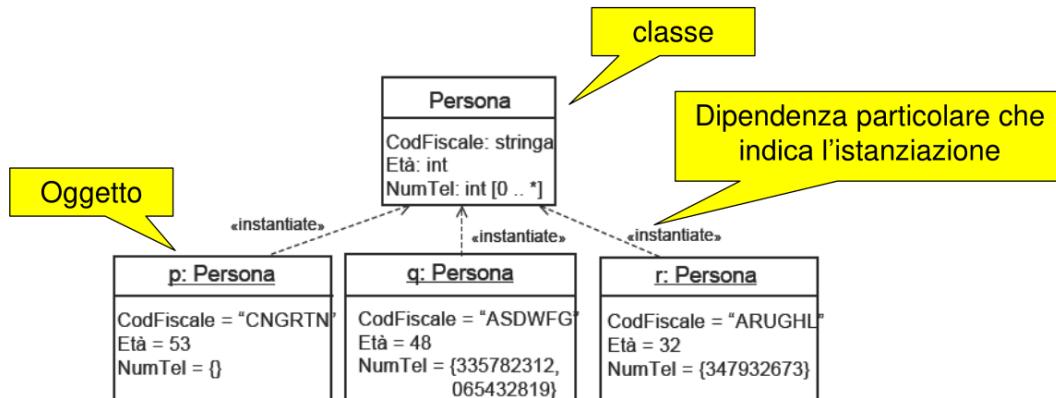
- È l'indicazione di quanti oggetti possono entrare a far parte di una proprietà
- Le molteplicità più comuni sono:
 - 1, 0..1, *
- In modo più generale, le molteplicità si possono definire indicando gli estremi inferiore e superiore di un intervallo (per esempio 2..4).
- Molti termini si riferiscono alla molteplicità degli attributi:
 - Opzionale: indica un limite inferiore di 0
 - Obbligatorio: implica un limite inferiore di 1 o più
 - A un solo valore: implica un limite superiore di 1
 - A più valori: implica che il limite superiore sia maggiore di 1, solitamente

1.7.2.2 Visibilità

- È possibile etichettare ogni operazione o attributo con un identificatore di visibilità
- UML fornisce comunque quattro abbreviazioni per indicare la visibilità:
 - + (public)
 - - (private)
 - ~ (package)
 - # (protected)

1.7.2.3 Attributi: Molteplicità

- Nelle istanze, il valore di un attributo multi-valore si indica mediante un insieme



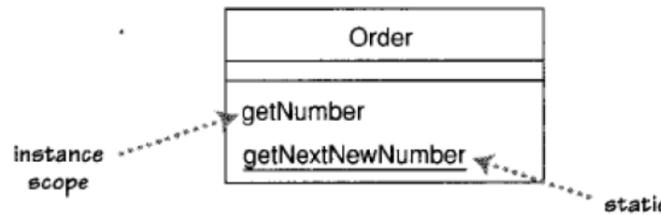
1.7.2.4 Operazioni

- Le operazioni sono le azioni che la classe sa eseguire, e in genere si fanno corrispondere direttamente ai metodi della corrispondente classe a livello implementativo

- Le operazioni che manipolano le proprietà della classe di solito si possono dedurre, per cui non sono incluse nel diagramma
- La sintassi UML completa delle operazioni è **visibilità nome (lista parametri) : tipo ritorno {stringa di propo}**
 - Visibilità: operazione pubblica (+) o privata (-)
 - Nome: stringa
 - Lista parametri: lista parametri dell'operazione
 - Tipo di ritorno: tipo di valore restituito dall'operazione, se esiste
 - Stringa di proprietà: caratteristiche aggiuntive che si applicano all'operazione

1.7.2.5 Operazioni e Attributi Statici

- UML chiama **static** un'operazione o un attributo che si applicano a una classe anziché alle sue istanze
- Questa definizione equivale a quella dei membri statici nei linguaggi come per esempio java e C
- Le caratteristiche statiche vanno sottolineate sul diagramma



1.7.2.6 Associazioni

- Le associazioni sono un altro modo di rappresentare le proprietà
- Gran parte dell'informazione che può essere associata a un attributo si applica anche alle associazioni
- Un'associazione è una linea continua che collega due classi, orientata dalla classe sorgente a quella destinazione
- Il nome e la molteplicità vanno indicati vicino all'estremità finale dell'associazione:
- la classe destinazione corrisponde al tipo della proprietà



- È possibile assegnare un nome all'associazione e anche assegnare dei nomi ai "ruoli" svolti da ciascun elemento di un associazione
- Gran parte delle cose ce valgono per gli attributi valgono anche per le associazioni
- Anche nei casi in cui non è strettamente necessario, il ruolo può essere utile per aumentare la leggibilità del diagramma
- Ovviamente quando abbiamo un associazione il tipo della proprietà corrisponde alle classi che sono dall'altra parte della associazione, cioè se guardiamo a la proprietà nella classe A sarà di tipo classe B, e viceversa se guardiamo a classe B la proprietà sarà di tipo classe A



1.7.2.7 Associazioni Bidirezionali

- Una tipologia di associazione è quella **bidirezionale (o binaria)**, costituita da una coppia di proprietà collegate, delle quali una è l'inversa dell'altra
- Il collegamento inverso implica che, se seguite il valore di una proprietà e poi il valore della proprietà collegata, dovrete ritornare all'interno di un insieme che contiene il vostro punto di partenza

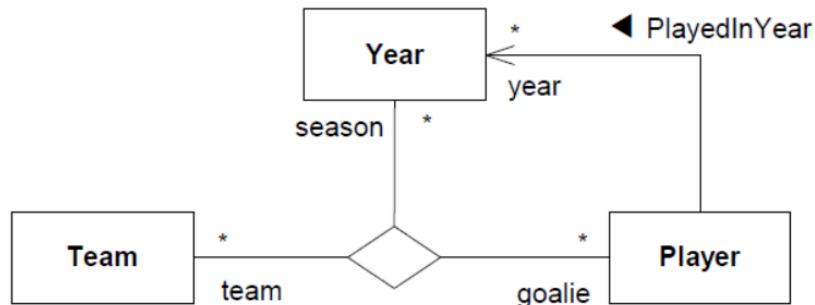


la natura bidirezionale dell'associazione è palesata dalle **frecce di navigabilità** aggiunte a entrambi i capi della linea

- Con questa specifica stiamo dicendo all'implementatore che nel progetto deve essere garantita la navigabilità in entrambi i sensi; nel caso non ci sia una specifica di navigabilità la realizzazione è deliberata all'implementatore

1.7.2.8 Associazioni Ternarie

Quando si hanno associazioni ternarie (o che coinvolgono più classi) si introduce il simbolo “**diamante**”



1.7.2.9 Associazioni: molteplicità

- La specifica UML (vista fino a ora) dichiara che la molteplicità di un'associazione è *the multiplicity of instances of that entity is the range of number of objects that participate in the association from the perspective of the other end (the other class)* in italiano

la molteplicità delle istanze di quell'entità è l'intervallo del numero di oggetti che partecipano all'associazione dal punto di vista dell'altro estremo (l'altra classe)

- Tale definizione (derivata dalla specifica E/R originale) non può però applicarsi alle associazioni multiple
- Pertanto, come già visto nel corso SIT, la notazione usata in questo corso (e in altri) prevede che

la molteplicità di un’associazione rappresenta il numero (minimo e massimo) di istanze dell’associazione a cui un’istanza dell’entità può partecipare

1.7.3 Classi di Associazione

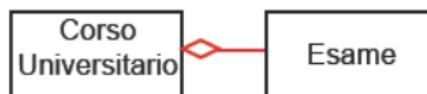
- In UML non si possono aggiungere attributi come in ER, ma si creano delle classi di associazione, che permettono di aggiungere attributi, operazioni e altre caratteristiche a una classe, che sono proprie dell’associazione
- In UML si indicano le classi di associazione con una linea tratteggiata, o utilizzando un rombo, come in figura



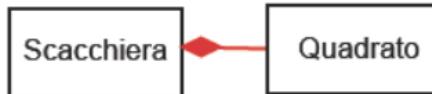
- In generale le classi di associazione sono delle classi che sono venute fuori più avanti nel progetto, perché, ad esempio, l’associazione in figura si può realizzare anche con una classe normale
- La classe di associazione aggiunge implicitamente un vincolo extra: ci può essere solo un’istanza della classe di associazione tra ogni coppia di oggetti associati

1.7.4 Aggregazione e Composizione

- **Aggregazione:**
 - è un associazione che corrisponde a una relazione intuitiva Intero-Parte (“**part-of**”)
 - è rappresentata da un **diamante vuoto** sull’associazione, posto vicino alla classe le cui istanze sono gli “interi”



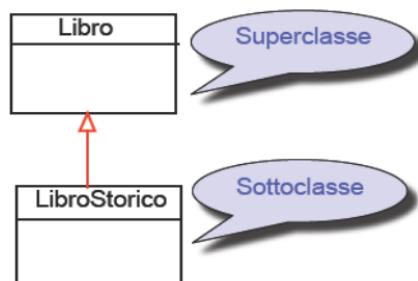
- è una relazione **binaria**
- può essere **ricorsiva**
- **Composizione:**
 - è un aggregazione che rispetta due vincoli ulteriori:
 - una parte può essere inclusa al massimo un intero in ogni istante
 - solo l’oggetto intero può creare e distruggere le sue parti, cioè, le parti non esistono al di fuori dell’intero; questo implica, nell’implementazione, che non esiste un costruttore per le parti
 - è rappresentata da un **diamante pieno** vicino alla classe che rappresenta gli “interi”



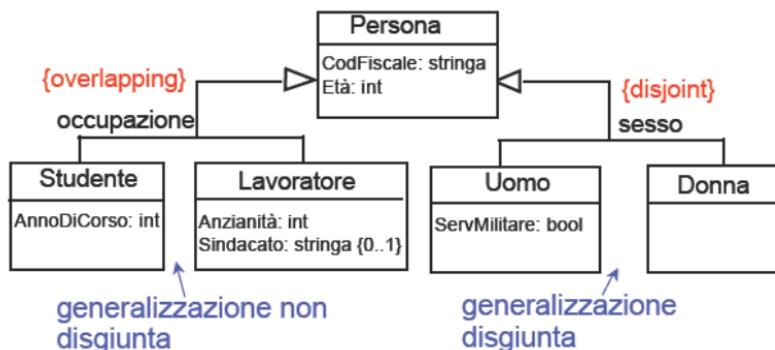
- se l'oggetto che compone viene distrutto, anche i figli vengono distrutti, anche se i figli possono essere creati/distrutti in momenti diversi dalla creazione/distruzione dell'oggetto che compone
- può essere **ricorsiva**

1.7.5 Generalizzazione

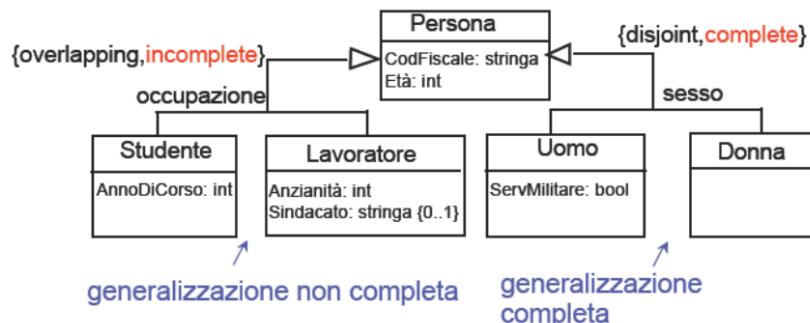
- La generalizzazione è indicata con una **freccia vuota** fra due classi dette **sottoclasse e superclasse**
- Il significato della generalizzazione è il seguente: ogni istanza della sottoclasse è anche istanza della superclasse



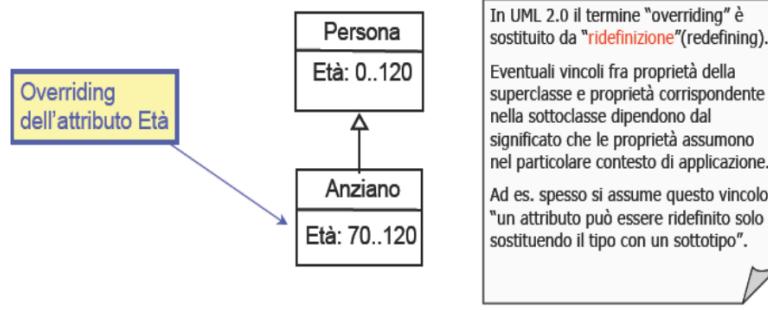
- La stessa superclasse può partecipare a diverse generalizzazioni
- Una generalizzazione può essere **disgiunta**, cioè le sottoclassi sono disgiunte (non hanno istanze in comune), o meno



- Una generalizzazione può essere **completa** (l'unione delle istanze delle sottoclassi è uguale all'insieme delle istanze della superclasse), o meno
- Attenzione:** i valori di default sono **{incomplete, disjoint}**

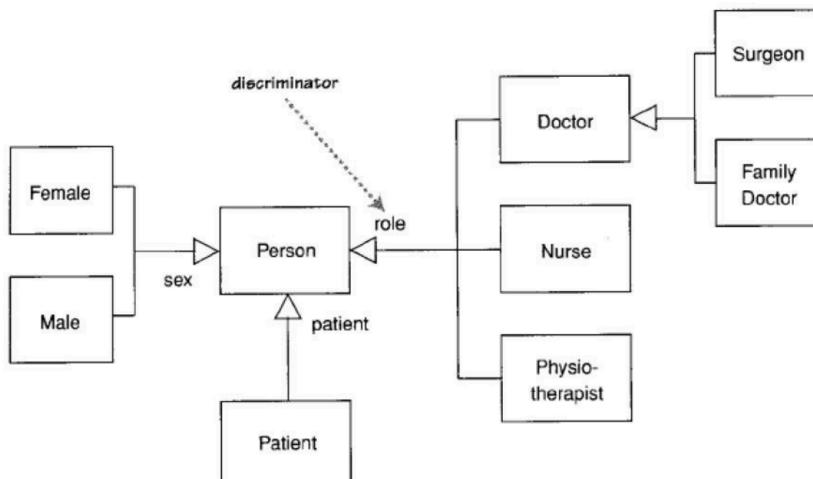


- In una generalizzazione la sottoclasse non solo può avere caratteristiche aggiuntive rispetto alla superclasse, ma può anche **sovrascrivere (overriding)** le proprietà ereditate dalla superclasse



1.7.6 Generalizzazione Multipla

- Con la **generalizzazione singola** un oggetto appartiene a un solo tipo, che può eventualmente ereditare dai suoi tipi padre
- Con la **generalizzazione multipla** un oggetto può essere descritto da più tipi, non necessariamente collegati dall'ereditarietà
- Si noti che la generalizzazione multipla è una cosa ben diversa dall'ereditarietà multipla
 - **Ereditarietà multipla:** un tipo può avere più supertipi, ma ogni oggetto deve sempre appartenere a un suo tipo ben definito
 - **Generalizzazione multipla:** un oggetto viene associato a più tipi senza che per questo debba esserne appositamente definito un altro; ad esempio, ogni istanza di persona può essere sia uomo sia studente
- Se si usa la generalizzazione multipla, ci si deve assicurare di rendere chiare le combinazioni "legali"
- Per questo fatto, UML pone ogni relazione di generalizzazione in un **insieme di generalizzazione**
- Sul diagramma delle classi, la freccia che indica una generalizzazione va etichettata con il nome del rispettivo insieme
- La generalizzazione singola corrisponde all'uso di un singolo anonimo insieme di generalizzazione
- Gli insiemi di generalizzazione sono disgiunti per definizione
 - Ogni istanza del supertipo può essere istanza di uno dei sottotipi all'interno di quel sottoinsieme

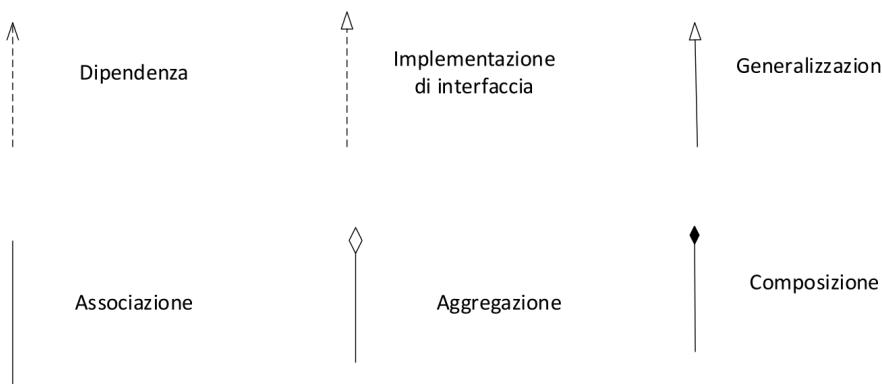


- Questa figura ci dice che ci sono tre modi diversi di vedere una persona:
 - dal punto di vista del sesso
 - dal punto di vista dell'essere paziente o meno

- dal punto di vista del ruolo

1.7.6.1 Relazione tra classi: sintassi

- Attenzione all'uso corretto delle frecce: UML è un linguaggio (anche se grafico) e scambiare una freccia per un'altra è un errore non da poco

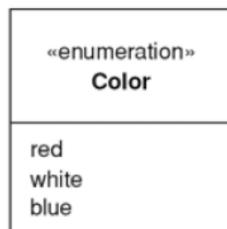


1.7.7 Classi Astratte

- Una **classe astratta** è una classe che non può essere direttamente istanziata: per farlo bisogna prima crearne una sottoclasse concreta
- Tipicamente, una classe astratta ha una o più operazioni astratte
- Un'operazione astratta non ha implementazione
 - è costituita dalla sola dichiarazione, resa pubblica affinché le classi client possano usufruirne
- Il modo più diffuso di indicare una classe o un'operazione astratta in UML è **scrivere il nome in corsivo**
- Si possono anche rendere astratte le proprietà indicandole direttamente come tali o rendendo astratti i metodi d'accesso
- A cosa servono?
 - servono come superclassi comuni per un insieme di sottoclassi concrete
 - queste sottoclassi, in virtù del subtyping, sono in qualche misura compatibili e intercambiabili fra di loro
 - infatti sono tutte sostituibili con la superclasse
 - per il principio di sostituzione di Liskov, tutte le istanze delle sottoclassi sono anche istanze della superclasse, quindi possono attuare qualsiasi comportamento descritto dalla superclasse, inclusi i comportamenti astratti

1.7.8 Enumerazioni

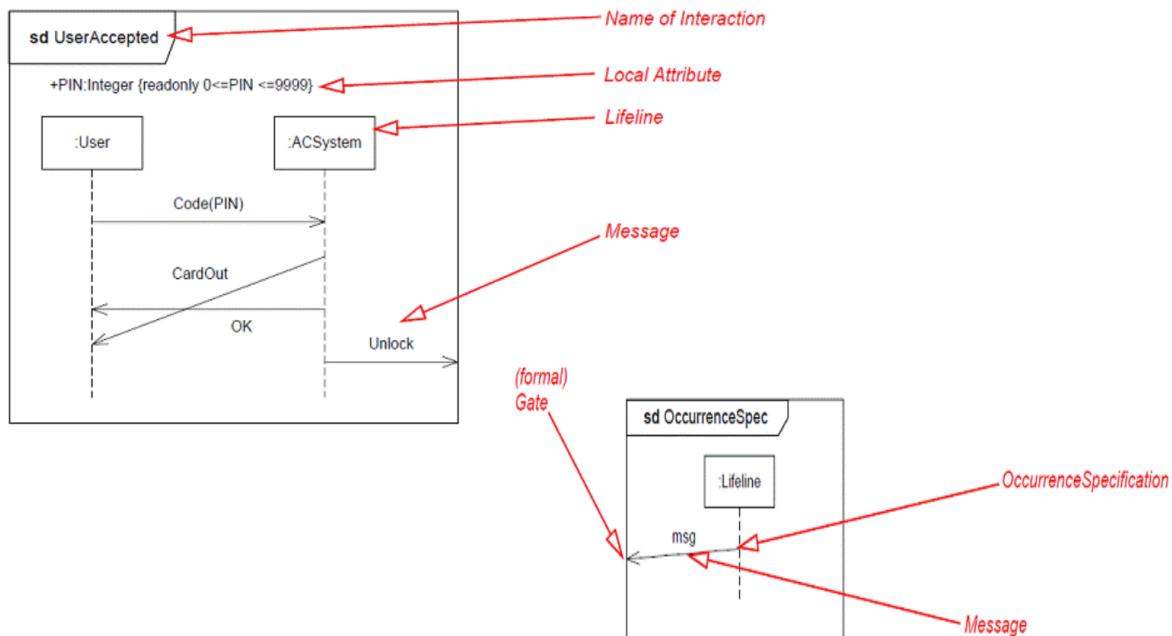
- Le enumerazioni sono usate per mostrare un insieme di valori prefissati (quelli scritti all'interno) che non hanno altre proprietà oltre al loro valore simbolico
- Sono rappresentate con una classe marcata dalla parola chiave «enumeration»



1.8 Diagramma di sequenza

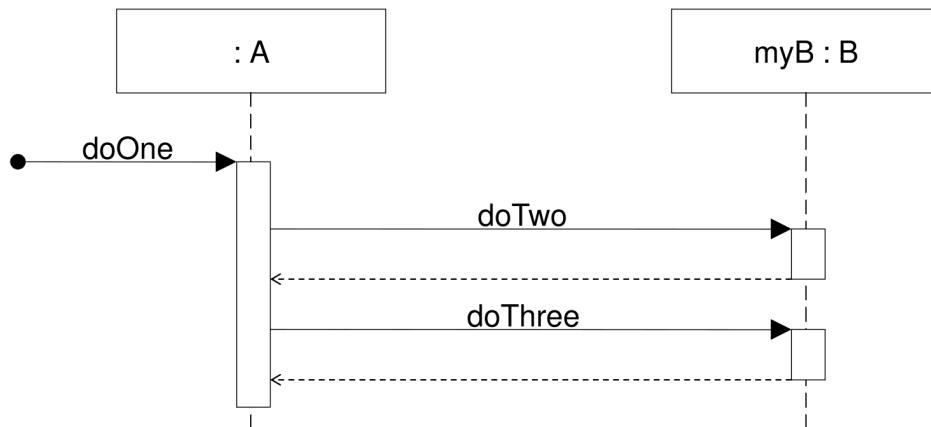
- Diagramma che illustra le interazioni tra le classi / entità disponendole lungo una sequenza temporale

- In particolare mostra i soggetti (chiamati tecnicamente **Lifeline**) che partecipano all'interazione e la sequenza dei messaggi scambiati
- In ascissa troviamo i diversi soggetti (non necessariamente in ordine di esecuzione), mentre in ordinata abbiamo la scala dei tempi sviluppata verso il basso



- Le linee della *Lifeline* partono da un quadrato, un'entità definita con sintassi UML. In questo caso i ":" indicano un'istanza della classe
- Tra la classe `User` e la classe `ACSystem` ci sarà sicuramente un'associazione
- La figura ci dice inoltre che i due messaggi che vengono inviati dopo la ricezione di PIN possono venire inviati in momenti diversi

1.8.1 Esempio



- La freccia con linea continua e piena indica un messaggio sincrono, il chiamante rimane in attesa della risposta del chiamato
- La freccia con linea tratteggiata e vuota è il messaggio che il chiamato invia al chiamante

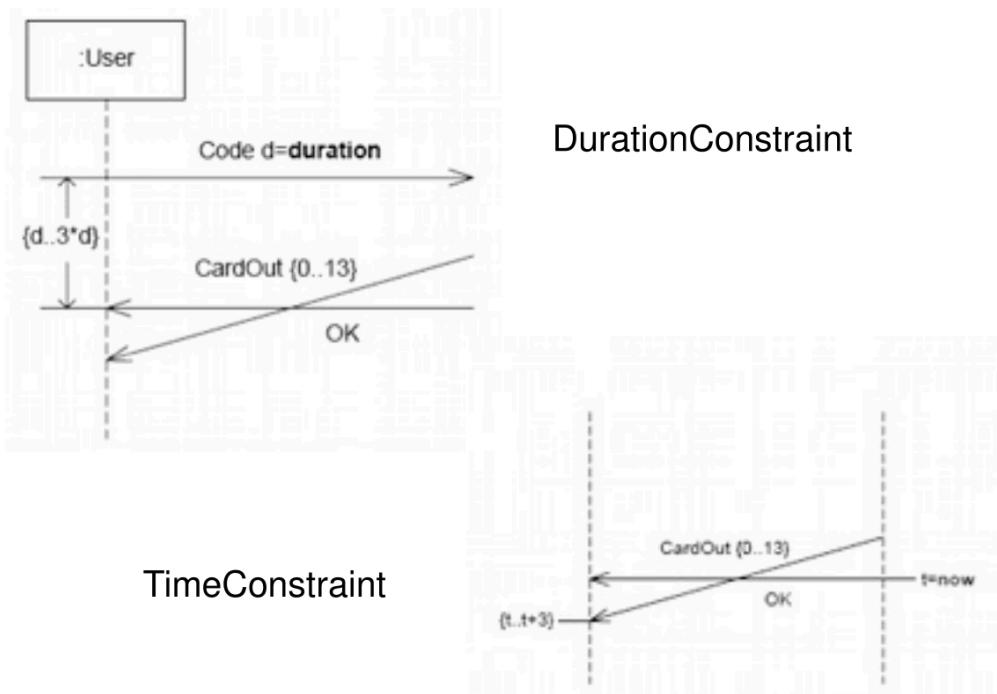
1.8.2 Lifeline

- In un diagramma di sequenza, i partecipanti solitamente sono istanze di classi UML caratterizzate da un nome
- La loro vita è rappresentata da una **Lifeline**, cioè una linea **tratteggiata verticale ed etichettata**, in modo che sia possibile comprendere a quale componente del sistema si riferisce

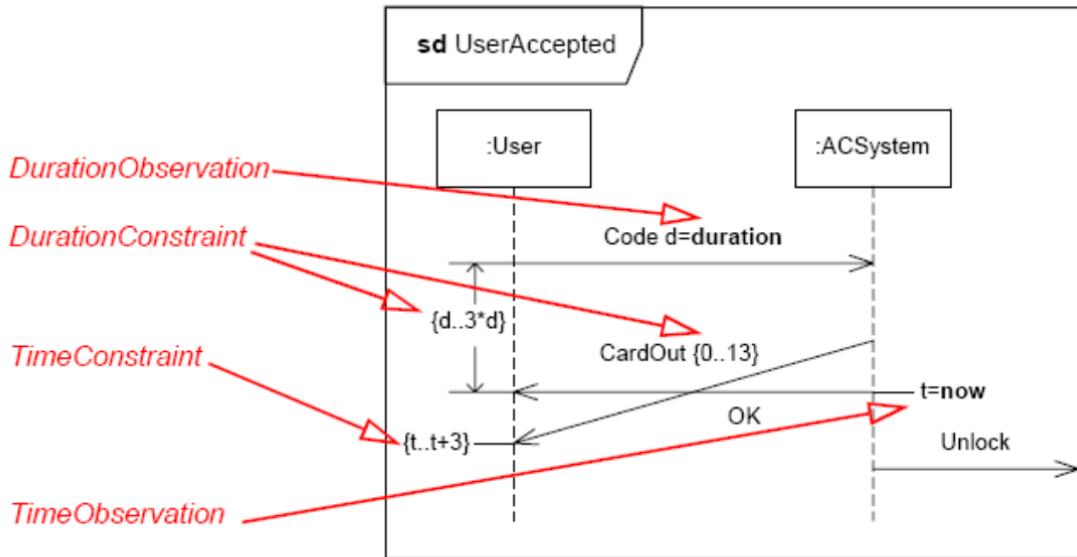
- In alcuni casi il partecipante non è un'entità semplice, ma composta
 - è possibile modellare la comunicazione fra più sottosistemi, assegnando una Lifeline ad ognuno di essi
- L'ordine in cui le OccurrenceSpecification (cioè l'invio e la ricezione di eventi) avvengono lungo la Lifeline **rappresenta esattamente l'ordine in cui tali eventi si devono verificare**
- La distanza (in termini grafici) tra due eventi non ha rilevanza dal punto di vista semantico
- Dal punto di vista notazionale, una Lifeline è rappresentata da un rettangolo che costituisce la “testa” seguito da una linea verticale che rappresenta il tempo di vita del partecipante
- È interessante notare che nella sezione della notazione, viene indicato espressamente che il “rettangolino” che viene apposto sulla Lifeline rappresenta l'attivazione di un metodo alla ricezione di un messaggio

1.8.2.1 Vincoli Temporali

- Per modellare sistemi real-time, o comunque qualsiasi altra tipologia di sistema in cui la temporizzazione è critica, è necessario specificare un istante in cui un messaggio deve essere inviato, oppure quanto tempo deve intercorrere fra un'interazione e un'altra
- Grazie, rispettivamente, a Time Constraint e Duration Constraint è possibile definire questo genere di vincoli

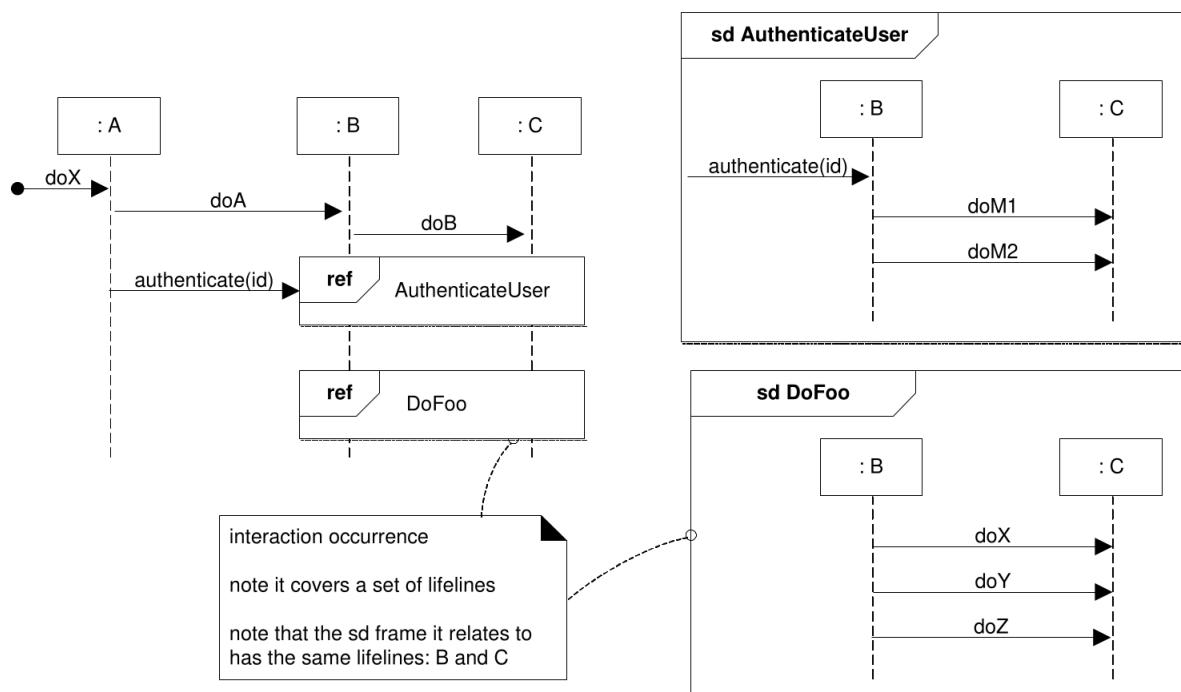


N.B. Noi non utilizzeremo mai vincoli temporali



1.8.3 Riferimento ad altri Diagrammi

- Spesso i diagrammi di sequenza possono assumere una certa complessità
 - necessità di poter definire comportamenti più articolati come composizione di nuclei di interazione più semplici
- Oppure, se una sequenza di eventi ricorre spesso, potrebbe essere utile definirla una volta e richiamarla dove necessario
- Per questa ragione, UML permette di inserire **riferimenti ad altri diagrammi** e passare loro degli argomenti
- Ovviamente ha senso sfruttare quest'ultima opzione solo se il diagramma accetta dei parametri sui quali calibrare l'evoluzione del sistema
- Questi riferimenti prendono il nome di **InteractionUse**
- I punti di connessione tra i due diagrammi prendono il nome di **Gate**
- Un Gate rappresenta un **punto di interconnessione** che mette in relazione un messaggio al di fuori del frammento di interazione con uno all'interno del frammento



1.8.3.1 Messaggio

- Un messaggio rappresenta un'interazione realizzata come comunicazione fra Lifeline
- Questa interazione può consistere nella creazione o distruzione di un'istanza, nell'invocazione di un'operazione, o nella emissione di un segnale
- UML permette di rappresentare tipi differenti di messaggi

1.8.3.1.1 Tipi di Messaggio

- Se sono specificati mittente e destinatario è un **complete message**
 - la semantica è rappresentata quindi dall'occorrenza della coppia di eventi `<sendEvent, receiveEvent>`
- Se il destinatario non è stato specificato è un **lost message**
 - in questo caso è noto solo l'evento di invio del messaggio
- Se il mittente non è stato specificato è un **found message**
 - in questo caso è noto solo l'evento di ricezione del messaggio
- Nel caso non sia noto né il destinatario né il mittente è un **unknown message**

NODE TYPE	NOTATION	REFERENCE
Message	<p>Code</p> <p>dolt(z)</p>	Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all <i>complete messages</i> .
Lost Message	<p>lost</p>	Lost messages are messages with known sender, but the reception of the message does not happen.
Found Message	<p>found</p>	Found messages are messages with known receiver, but the sending of the message is not described within the specification.
GeneralOrdering		

- Attenzione alle frecce che usate nei messaggi, hanno significati diversi:
 - **riga continua freccia piena:** indica un messaggio (**call**) **sincrono** in cui il mittente **aspetta** il completamento dell'esecuzione del destinatario prima di continuare la sua esecuzione

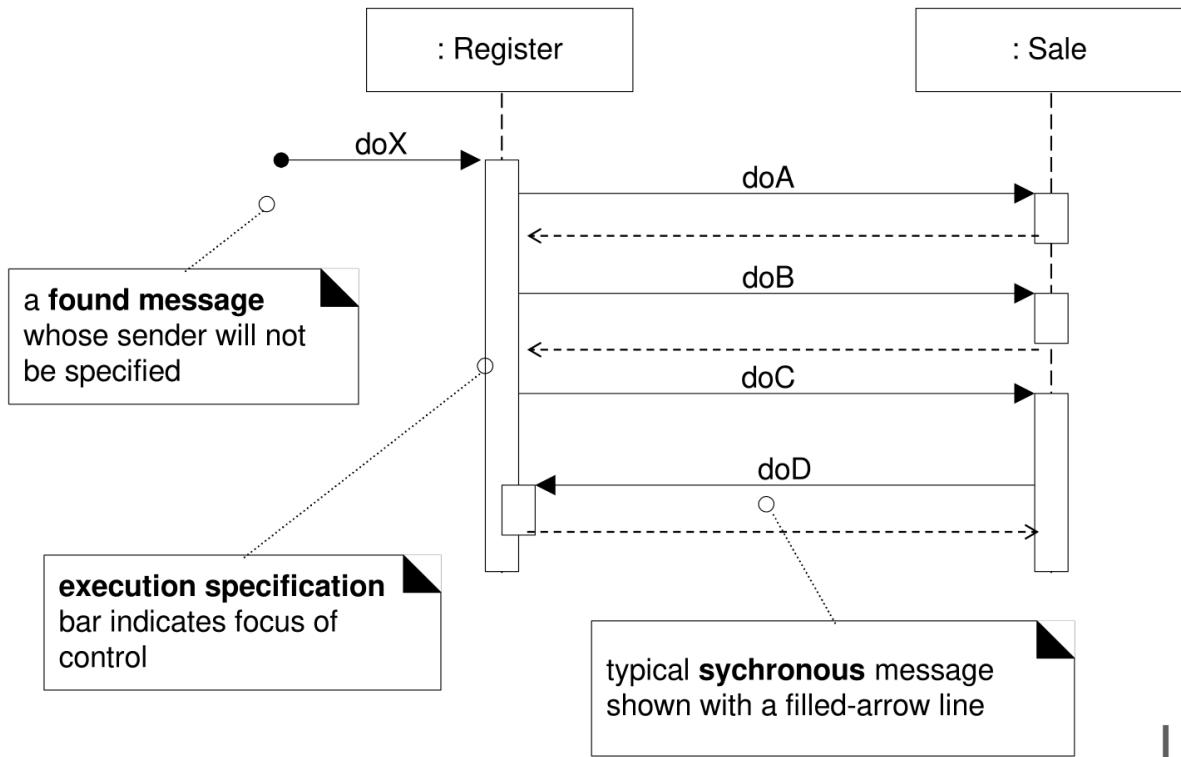


- Necessita di un messaggio di ritorno per sbloccare l'esecuzione del mittente
- **riga continua freccia vuota:** indica un messaggio **asincrono** in cui il mittente **non aspetta** il completamento dell'esecuzione del destinatario ma continua la sua esecuzione

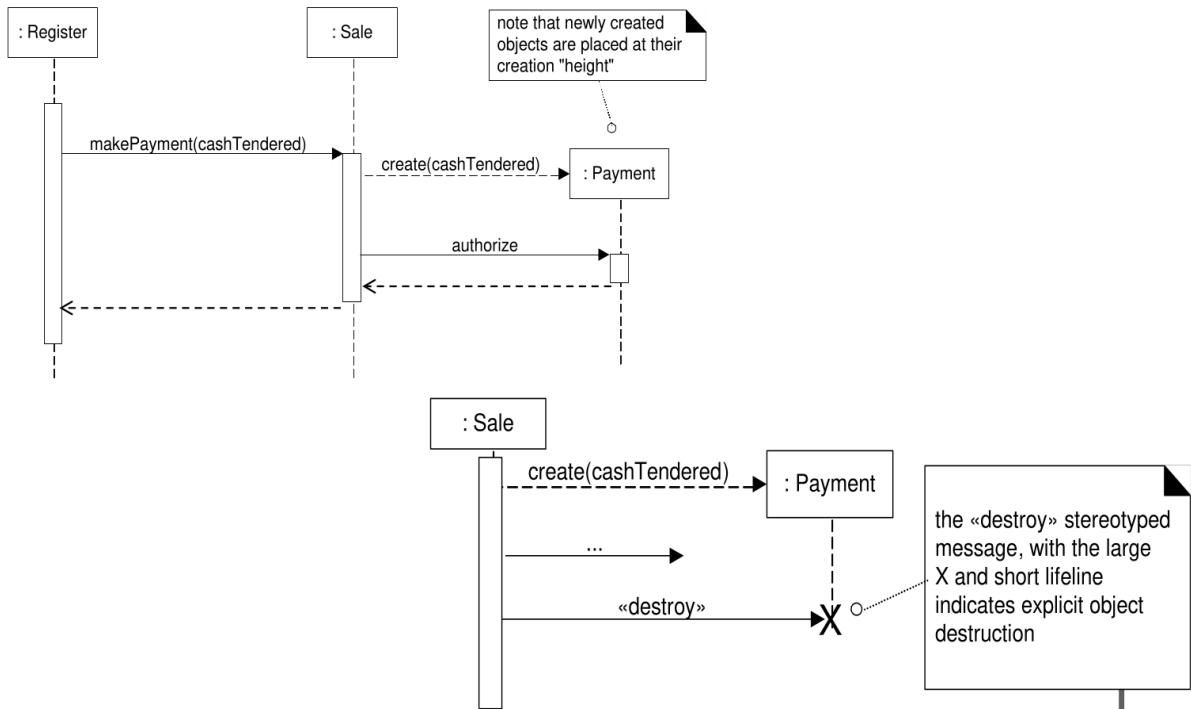


- Il valore di ritorno potrebbe o meno essere necessario, dipende dalla semantica
- **riga tratteggiata freccia vuota:** indica il ritorno di un messaggio





- Non specificchiamo chi è che ha inviato il messaggio, in questo diagramma non ci interessa
- Register manda il messaggio **doA** all'oggetto **Sale** e aspetta la sua terminazione
- Poi manda il messaggio **doB** all'oggetto **Sale** e aspetta la sua terminazione
- Infine invia il messaggio **doC**, e come conseguenza **Sale** invia **doD** alla classe **Register**



- **authorize**
 - Per realizzare **makePayment(cashTendered)** la classe **Sale** delega la classe **Payment** (creando un'istanza della classe stessa), che invoca il metodo **create(cashTendered)**; dopodiché invia il messaggio sincrono **authorize**, e restituisce il controllo a **Sale**
 - A questo punto **Sale** può notificare l'avvenuto pagamento
- **destroy**
 - con il messaggio **destroy** viene distrutta l'istanza di quella classe

- siccome siamo in *modellazione*, non dobbiamo soffermarci sul fatto che il nostro linguaggio di programmazione possa o meno poter distruggere un'istanza di una classe

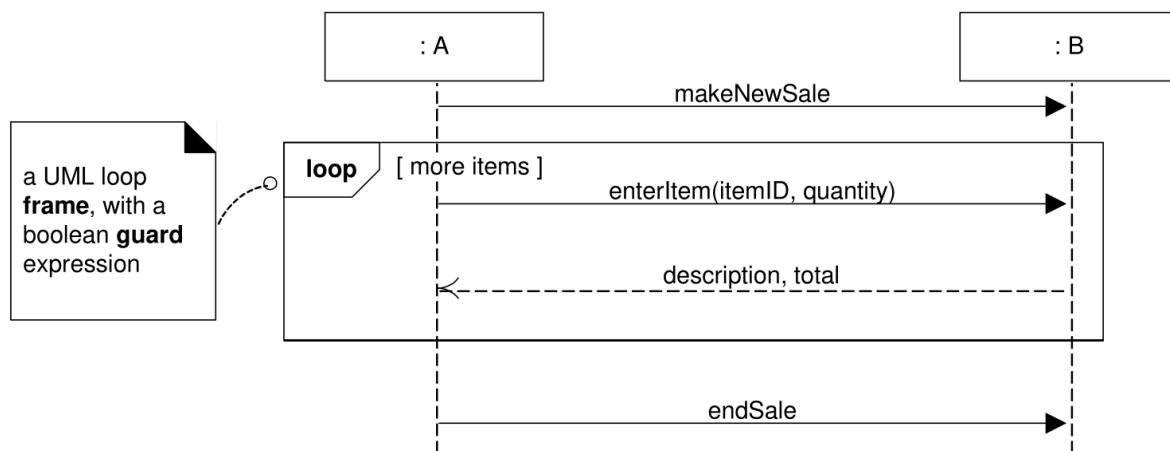
1.8.4 Combined Fragment

- La specifica di UML permette di esprimere comportamenti più complessi rispetto al singolo scambio di messaggi
- È possibile rappresentare l'esecuzione atomica di una serie di interazioni, oppure che un messaggio debba essere inviato solo in determinate condizioni
- A tale scopo UML mette a disposizione i *Combined Fragment*, cioè contenitori atti a delimitare un'area d'interesse nel diagramma
- Servono per spiegare che una certa catena di eventi, racchiusa in uno o più operandi, si verificherà in base alla semantica dell'operatore associato
- Ogni fragment ha un operatore e una (eventuale) guardia

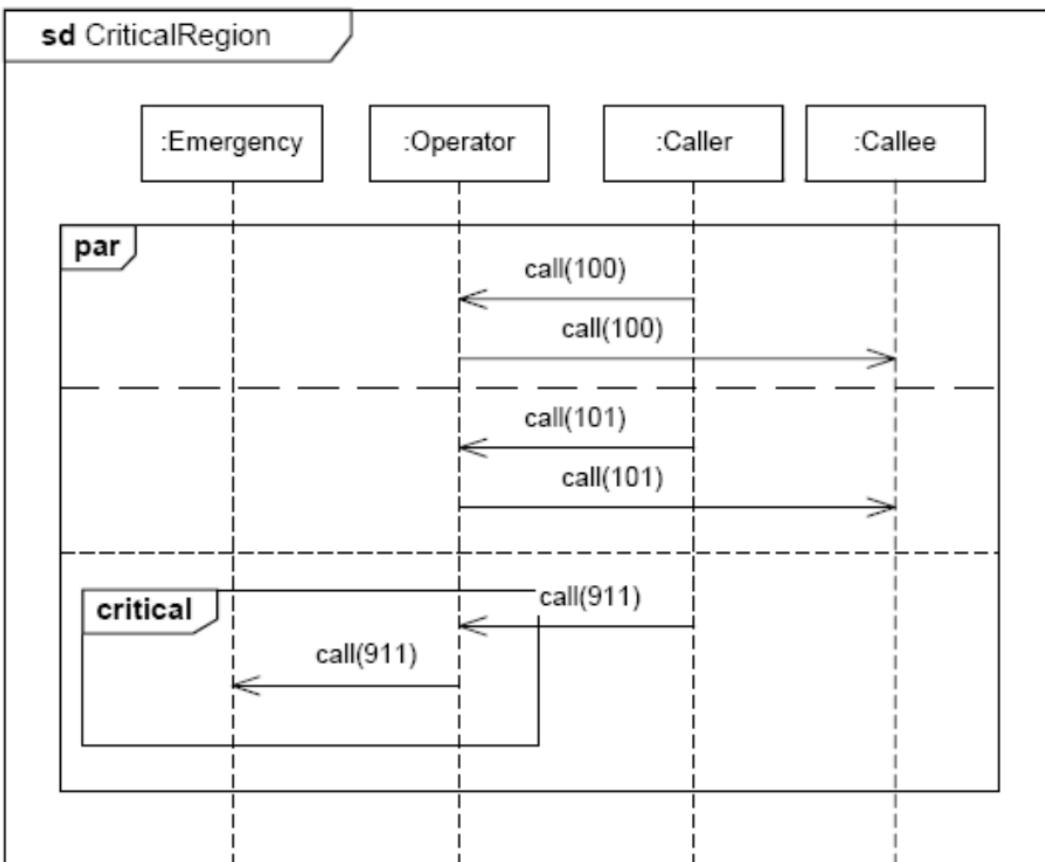
1.8.4.1 Tipi

- **Loop**: specifica che quello che è racchiuso nell'operando sarà eseguito ciclicamente finché la guardia sarà verificata
- **Alternatives** (alt): indica che sarà eseguito il contenuto di uno solo degli operandi, quello la cui guardia risulta verificata
- **Optional** (opt): indica che l'esecuzione del contenuto dell'operando sarà eseguita solo se la guardia è verificata
- **Break** (break): ha la stessa semantica di opt, con la differenza che in seguito l'interazione sarà terminata
- **Critical**: specifica un blocco di esecuzione atomico (non interrompibile)
- **Parallel** (par): specifica che il contenuto del primo operando può essere eseguito in parallelo a quello del secondo
- **Weak Sequencing** (seq): specifica che il risultato complessivo può essere una qualsiasi combinazione delle interazioni contenute negli operandi, purché:
 - l'ordinamento stabilito in ciascun operando sia mantenuto nel complesso
 - eventi che riguardano gli stessi destinatari devono rispettare anche l'ordine degli operandi, cioè i messaggi del primo operando hanno precedenza su quelli del secondo
 - eventi che riguardano destinatari differenti non hanno vincoli di precedenza vicendevole
- **StrictSequencing** (strict): indica che il contenuto deve essere eseguito nell'ordine in cui è specificato, anche rispetto agli operandi
- **Ignore**: indica che alcuni messaggi, importanti ai fini del funzionamento del sistema, non sono stati rappresentati, perché non utili ai fini della comprensione dell'interazione
- **Consider**: è complementare ad ignore
- **Negative** (neg): racchiude una sequenza di eventi che non deve mai verificarsi
- **Assertion** (assert): racchiude quella che è considerata l'unica sequenza di eventi valida
 - di solito è associata all'utilizzo di uno State Invariant come rinforzo

NODE TYPE	NOTATION	REFERENCE
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner.
CombinedFragment		



Fintanto che ci sono degli item, io faccio enter item e quello mi restituisce description total. Quando non esistono più item invio il messaggio endSale.

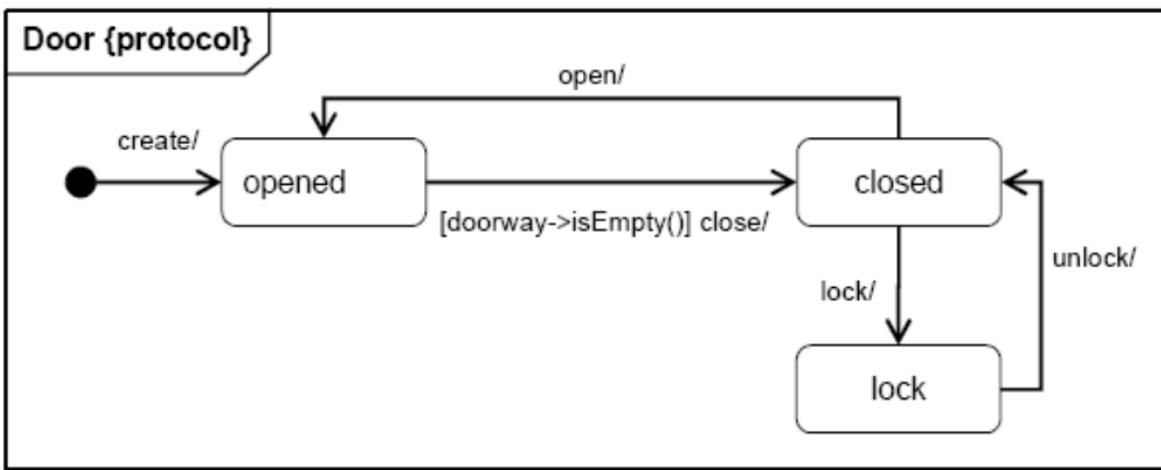


Fintanto che il chiamante invoca certi numeri, ridireziona quei numeri al chiamato, se viene chiamato il 991 quella è una chiamata critica che non può essere interrotta.

1.9 Diagrammi di Stato

- I **diagrammi di stato** modellano la dipendenza che esiste tra lo stato di una classe / entità (cioè il valore corrente delle sue proprietà) e i messaggi e/o eventi che questo riceve in ingresso
- Specifica il ciclo di vita di una classe / entità, definendo le regole che lo governano
- Quando una classe / entità si trova in un certo stato può essere interessata a determinati eventi (e non ad altri)
- Come risultato di un evento una classe / entità può passare a un nuovo stato (transizione)
- Uno **stato** è una condizione o situazione nella vita di un oggetto in cui esso soddisfa una condizione, esegue un'attività o aspetta un evento
- Un **evento** è la specifica di un'occorrenza che ha una collocazione nel tempo e nello spazio
- Una **transizione** è il passaggio da uno stato a un altro in risposta ad un evento

1.9.1.1 Esempio

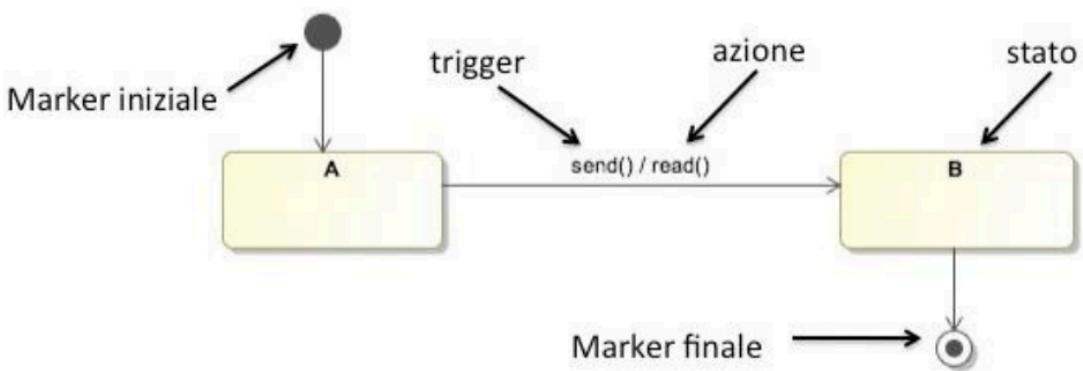


- Quando la porta è creata, la porta è aperta.
- Fintanto che la porta è aperta, è interessata a un unico messaggio: `close`; tutti gli altri messaggi vengono ignorati
- Quando riceve il messaggio `close`, e, parentesi quadra che indica la guardia, `doorway -> isEmpty()`, cioè non c'è nessuno sulla soglia, la porta transita dallo stato `open` allo stato `closed`
- Dunque la porta si potrebbe trovare in un stato `closed`, ed è interessata a due messaggi:
 - il messaggio `open`, che la riporta nello stato `open`
 - il messaggio `lock`, che la porta nello stato `lock`
- Quando si trova nello stato `lock`, la porta può essere interessata solo all'evento `unlock`, che la riporta nello stato `closed`

1.9.1.2 Concetti

- I concetti più importanti di un diagramma di stato sono:
 - gli **stati**, indicati con rettangoli con angoli arrotondati
 - le **transizioni** tra stati, indicate attraverso frecce
 - gli **eventi** che causano transizioni, la tipologia più comune è rappresentata dalla ricezione di un messaggio, che si indica semplicemente scrivendo il nome del messaggio con relativi argomenti vicino alla freccia; quindi l'oggetto, l'istanza, riceve un messaggio, e ricevendo quel messaggio effettua la transizione
 - i **marker** di inizio e fine rappresentati rispettivamente da un cerchio nero con una freccia che punta allo stato iniziale e come un cerchio nero racchiuso da un anello sottile
 - le **azioni** che una entità è in grado di eseguire in risposta alla ricezione di un evento; cioè, se ricevo quel evento, oltre a fare la transizione, posso fare anche altre cose
 - il **vertice** che rappresenta l'astrazione di nodo nel diagramma e può essere la sorgente o la destinazione di una o più transizioni; rappresenta un sotto-diagramma

1.9.1.3 Esempio

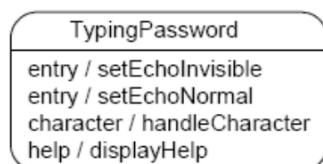


- Marker iniziale, stato iniziale, a seguito di un messaggio ricevuto, effetto quest'azione e vado in transizione verso un nuovo stato
- Da quello stato, quando posso, quando ricevo la distruzione, vado al marker finale

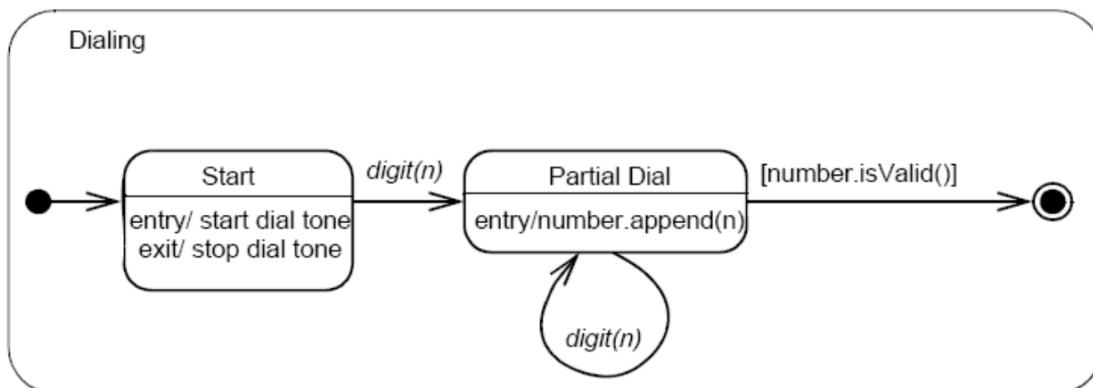
1.9.1.4 Stato

- Uno stato modella una situazione durante la quale vale una condizione invariante (solitamente implicita)
- L'invariante può rappresentare una situazione statica, come un oggetto in attesa che si verifichi un evento esterno
- Tuttavia, può anche modellare condizioni dinamiche, come il processo di esecuzione di un comportamento (cioè, l'elemento del modello in esame entra nello stato quando il comportamento inizia e lo lascia non appena il comportamento è completato)
- Stati possibili:
 - [Simple state](#)
 - [Composite state](#)
 - [Submachine state](#)
- Composite state:
 - può contenere una regione oppure è decomposto in due o più regioni ortogonali
 - ogni regione ha il suo insieme di sotto-vertici mutuamente esclusivi e disgiunti e il proprio insieme di transizioni
 - ogni stato appartenente ad una regione è chiamato substato
 - la transizione verso il marker finale all'interno di una regione rappresenta il completamento del comportamento di quella regione
 - quando tutte le regioni ortogonali hanno completato il loro comportamento questo rappresenta il completamento del comportamento dell'intero stato e fa scattare il trigger dell'evento di completamento
- Simple state:
 - è uno stato che non ha sottostati
- Submachine state:
 - specifica l'inserimento di un diagramma di una sotto-parte del sistema e permette la fattorizzazione di comportamenti comuni e il riuso dei medesimi
 - è semanticamente equivalente a un composite state, quindi le regioni del submachinestate sono come le regioni del composite state
 - le azioni sono definite come parte dello stato
- Uno stato può essere ridefinito:
 - uno stato semplice può essere ridefinito diventando uno stato composito
 - uno stato composito può essere ridefinito aggiungendo regioni, vertici, stati, transizioni e azioni

1.9.1.5 Esempio



Simple state con azioni

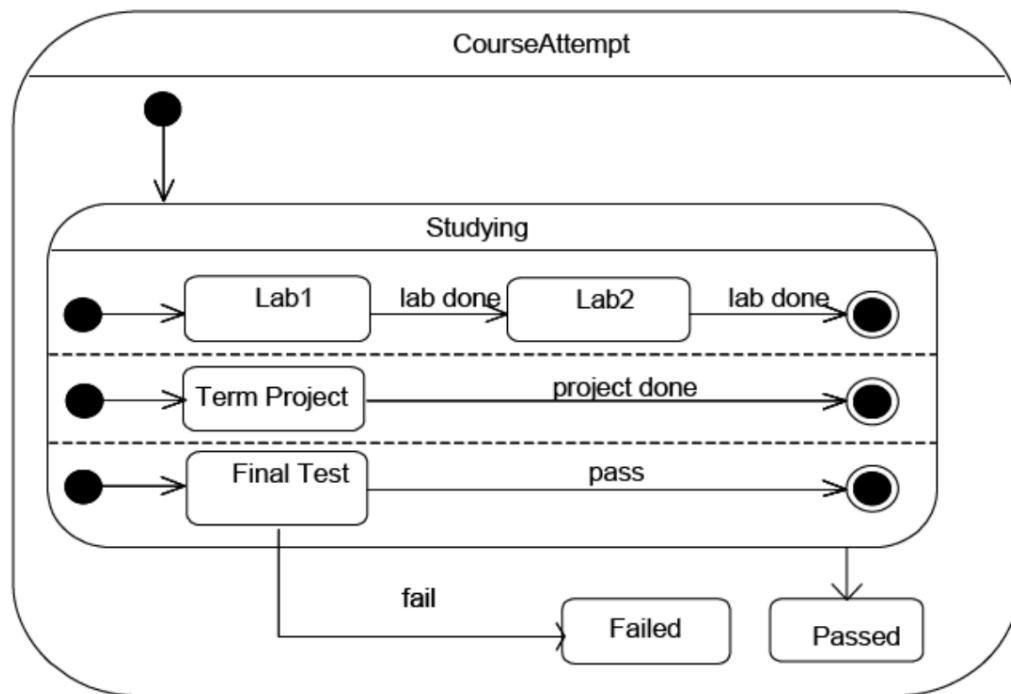


Composite state

Questo è uno stato composito, nel senso che quello stato composito può essere inserito all'interno di un altro diagramma di stato. E quindi rappresenta la fattorizzazione che abbiamo visto, è analoga a quello che abbiamo visto nel diagramma dell'attività.

1.9.1.6 Esempio

Composite state con regioni ortogonali

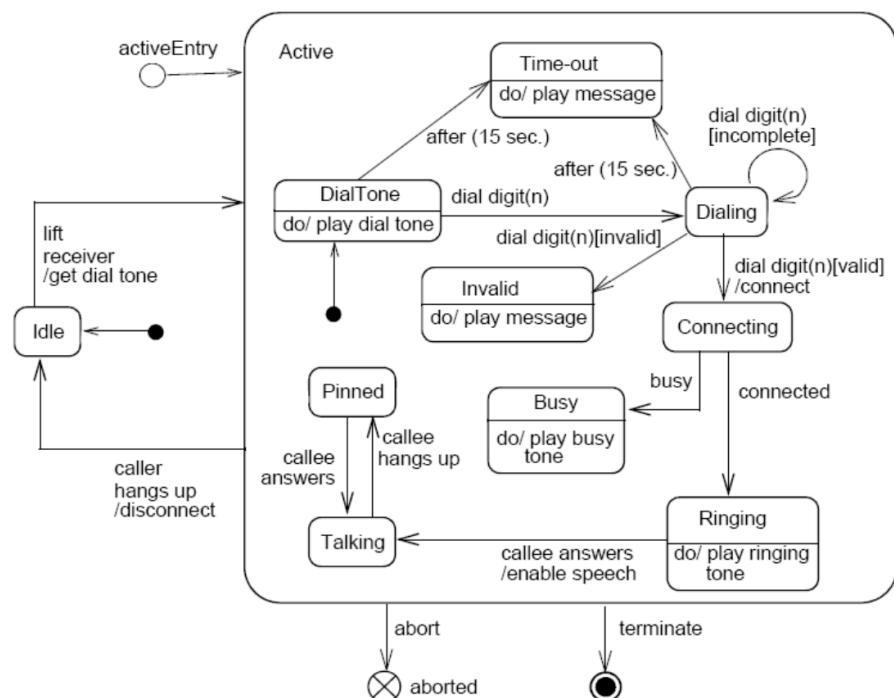


- Una Regione è una parte ortogonale di uno stato composto o di un state machine
- Questo è uno stato composito con regioni ortogonali che rappresenta il diagramma di stato per il superamento di un esame da parte dello studente

- In questo stato di piano ci sono le tre regioni che sono eseguite “concorrentemente”, possono essere eseguite in paralleli, non sto indicando una sequenzialità
- Possono iniziare contemporaneamente ma non finiscono per forza nello stesso istante
- Il diagramma dice che il design prevede lo svolgimento della Lab1, dal quale si esce quando lab done
- A quel punto posso fare Lab2, quando ho fatto Lab2, lab done, ho terminato quella regione
- Quindi si termina quella regione quando ho terminato Lab1 e Lab2
- Nella seconda regione si esce quando il progetto è terminato
- In fine c'è il final test; dal final test si può uscire con due messaggi
 - fail che provoca automaticamente la segnalazione dell'esame come fallito, Failed
 - pass, allora termine quella regione, e l'esame viene segnato come passato, Passed

1.9.1.7 Esempio

Submachine state



1.9.1.8 Pseudostato

- Gli pseudostati vengono generalmente utilizzati per collegare più transizioni in percorsi di transizioni di stato più complessi
- Ad esempio, combinando una transizione che entra in uno pseudostato fork con un insieme di transizioni che escono dallo pseudostato fork, otteniamo una transizione composta che porta a un insieme di stati ortogonali

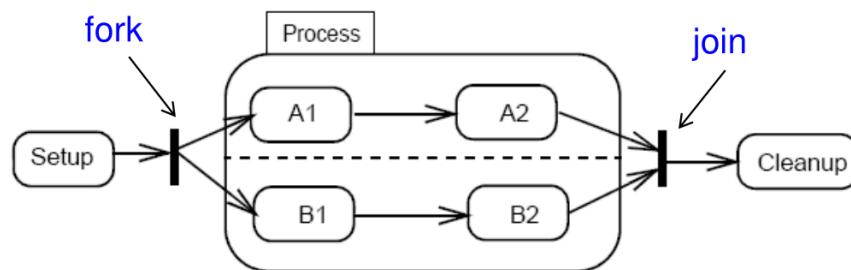
1.9.1.9 Tipi di Pseudostati

- **Initial:** vertice di default che è la sorgente della singola transizione verso lo stato di default di un composite state - Ci può essere al massimo un vertice initial e la transizione uscente da questo vertice non può avere trigger o guardie
- **deepHistory:** la più recente configurazione attiva del composite state che contiene direttamente questo pseudostato

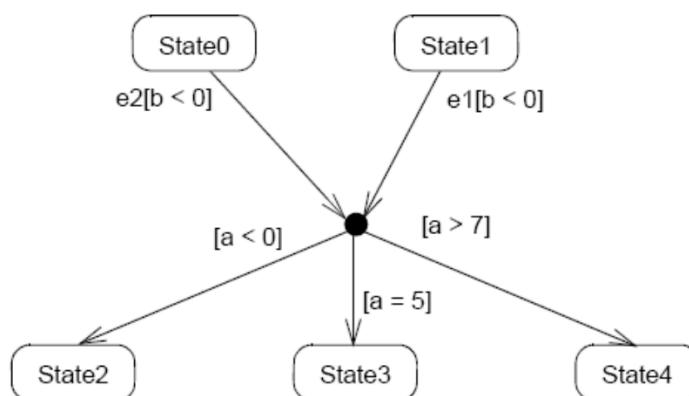
- **shallowHistory**: il più recente substate attivo di un composite state initial deepHistory shallowHistory



- **Join**: permette di eseguire il merge di diverse transizioni provenienti da diverse sorgenti appartenenti a differenti regioni ortogonalali
 - ▶ Le transizioni entranti in questo vertice non possono avere guardie e trigger
- **Fork**: permette di separare una transizione entrante in due o più transizioni che terminano in vertici di regioni ortogonalali
 - ▶ Le transizioni uscenti da questo vertice non possono avere guardie

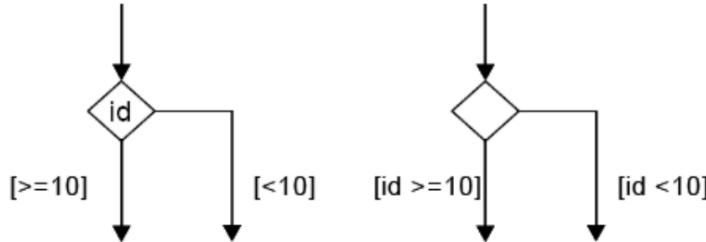


- **Junction**: è un vertice privo di semantica che viene usato per “incatenare” insieme transizioni multiple
 - ▶ È usato per costruire percorsi di transizione composti tra stati
 - ▶ Una junction può essere usata per far convergere transizioni multiple entranti in una singola transizione uscente che rappresenta un percorso condiviso di transizione
 - ▶ Allo stesso modo una junction può essere usata anche per suddividere una transizione entrante in più transizioni uscenti con differenti guardie
 - ▶ Permette di realizzare un branch condizionale statico, quindi quando esco da un certo stato, so già in quale delle biforcati finirò
 - ▶ Esiste una guardia predefinita chiamata “else” che viene attivata quando tutte le guardie sono false

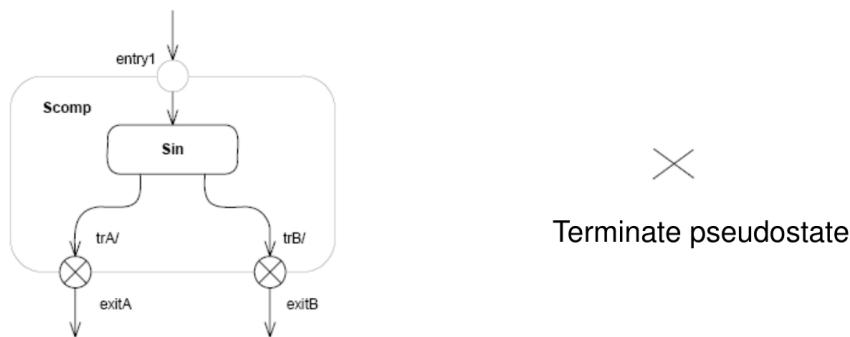


- **Choice**: è un tipo di vertice che quando viene raggiunto causa la valutazione dinamica delle guardie dei trigger delle transizioni uscenti
 - ▶ Le guardie sono quindi tipicamente scritte sotto forma di “funzione” che viene valutata al momento del raggiungimento del vertice choice
 - ▶ Permette di realizzare un branch condizionale dinamico separando le transizioni uscenti e creando diversi percorsi di uscita

- Se più di una guardia è verificata viene scelto il percorso in modo arbitrario
- Se nessuna è verificata il modello viene considerato “ill-formed”
- È quindi caldamente consigliato definire sempre un percorso di uscita di tipo “else”



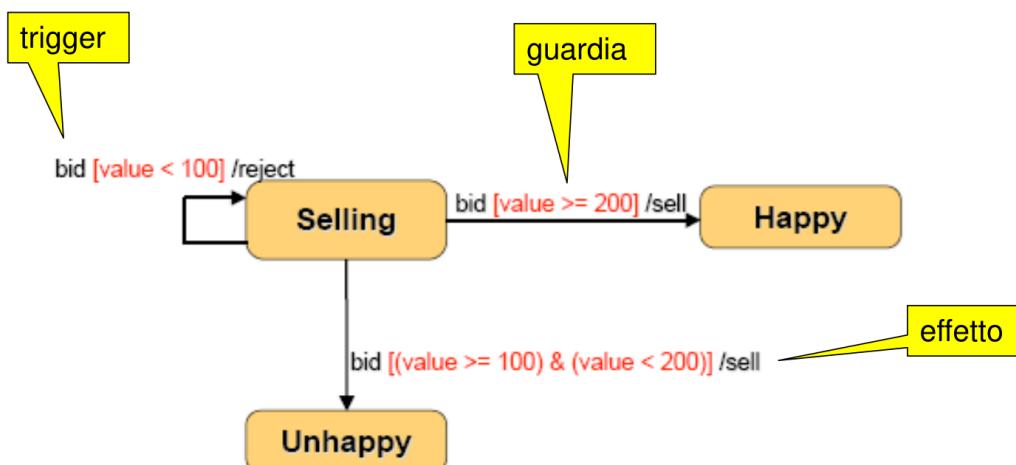
- La differenza tra *Junciton* e *Choice* è che, siccome nelle transizioni io posso eseguire delle azioni, la biforcazione di junction non può dipendere dalle azioni, mentre la biforcazione di choice può dipendere dall'esito di quelle azioni.
- **Entry point:** è l'ingresso di uno state machine o di un composite state
- **Exit point:** è l'uscita da uno state machine o da uno stato composito
 - Entrare in questo vertice significa attivare la transizione che ha questo vertice come sorgente
- **Terminate:** entrare in questo vertice implica che l'esecuzione di questo state machine è terminata
 - Lo state machine non uscirà da nessuno stato né verranno invocate altre azioni a parte quelle associate con la transizione che porta allo stato terminate



1.9.1.10 Transizione

- Una transizione è una relazione diretta tra un vertice di origine e un vertice di destinazione, un arco direzionale
- Può essere parte di una transizione composta, che porta la macchina a stati da una configurazione di stato a un'altra, rappresentando la risposta completa della macchina a stati al verificarsi di un evento di un tipo particolare; cioè, c'è un evento, definisco qual è la risposta della macchina a quell'evento, la risposta dell'istanza a quell'evento
- Analizzando la specifica UML è possibile vedere che tra le proprietà di una transizione compaiono diversi concetti molto rilevanti tra cui:
 - **trigger:** cioè i tipi di evento che possono innescare la transizione
 - **guardia:** cioè un vincolo che fornisce un controllo fine sull'innesto della transizione

- La guardia è valutata quando una occorrenza dell'evento è consegnata allo stato
 - Se la guardia risulta verificata allora la transizione può essere abilitata altrimenti questa viene disabilitata
 - Il problema delle guardie è che, in teoria, la guardia potrebbe essere rappresentata dalla esecuzione di operazione; se la guardia (l'operazione) ha degli effetti collaterali (*side effect*), potrebbe succedere il caso in cui la prima ricezione di un messaggio non blocca la transizione, mentre la seconda ricezione dello stesso messaggio blocca la transizione
 - Quindi le guardie dovrebbero essere pure espressioni senza *side effect*, i quali sono assolutamente sconsigliati nella specifica
- **effetto:** specifica un comportamento opzionale (cioè un'azione) che deve essere eseguito quando la transizione scatta trigger guardia effetto (l'effetto viene considerato da choose e non da junction)



- Sono in stato Selling, ricevo bid, scatta la guardia; se il valore è minore di 100 eseguo l'operazione reject, e rimango nello stato selling
- Se il valore è maggiore o uguale a 100 e minore di 200 alla ricezione del messaggio bid, eseguo l'operazione sell e passo in un stato Unhappy
- Se il valore è maggiore o uguale a 200 all'atto della ricezione del messaggio bid, eseguo l'operazione sell e passo nello stato Happy

1.9.1.11 Tipi di Eventi

- **Evento di chiamata:** ricezione di un messaggio che richiede l'esecuzione di una operazione
- **Evento di cambiamento:** si verifica quando una condizione passa da "falsa" a "vera"
 - Tale evento si specifica scrivendo <<when>> seguito da un'espressione, racchiusa tra parentesi tonde, che descrive la condizione
 - Utile per descrivere la situazione in cui un oggetto cambia stato perché il valore dei suoi attributi è modificato dalla risposta a un messaggio inviato
- Evento segnale: consiste nella ricezione di un segnale
- Evento temporale: espressione che denota un lasso di tempo che deve trascorrere dopo un evento dotato di nome
 - Con <> si può far riferimento all'istante in cui l'oggetto è entrato nello stato corrente
 - Con <> si esprime qualcosa che deve accadere in un particolare momento

1.9.1.12 Azione

- Un'azione è un elemento avente un nome che è l'unità fondamentale della funzionalità eseguibile
- L'esecuzione di un'azione rappresenta una trasformazione o elaborazione nel sistema modellato, sia esso un sistema informatico o altro
- **Entry:** tutte le volte che si entra in uno stato viene generato un evento di entrata a cui può essere associato uno o più specifici comportamenti che vengono eseguiti prima che qualsiasi altra azione possa essere eseguita
- **Exit:** tutte le volte che si esce da uno stato viene generato un evento di uscita a cui può essere associato uno o più specifici comportamenti che vengono eseguiti come ultimo passo prima che lo stato venga lasciato
- **Do:** rappresenta il comportamento che viene eseguito all'interno dello stato
 - il comportamento parte subito dopo l'ingresso nello stato (dopo che è stato eseguito l'entry)
 - se questo comportamento termina mentre lo stato è ancora attivo viene generato un evento di completamento e nel caso sia stata specificata una transizione per il completamento allora viene eseguito l'exit e successivamente la transizione d'uscita
 - se invece viene innescata una transizione di uscita prima che il do termini, allora l'esecuzione del do è abortita, viene eseguito l'exit e poi la transizione
- Il comportamento di entry è stato definito per **permettere di fattorizzare comportamenti comuni** all'ingresso di uno stato, sui quali non possiamo mettere la guardia
- Senza la possibilità di poter specificare un comportamento di entry, il progettista avrebbe dovuto specificare su ogni transizione verso lo stato una azione effetto legata alle transizioni
- Il medesimo discorso vale per le exit, nella quale sono fattorizzati tutti quei comportamenti che devono essere eseguiti prima di uscire dallo stato
- È importante però ricordare che non vi è la possibilità di esprimere condizioni di guardia su questi comportamenti

1.9.1.13 Diagramma di Stato

- Transizione interna che esclude azioni di entrata e di uscita



- Attività interna: generazione di un thread concorrente che resta in esecuzione finché:
 - il thread (eventualmente) termina
 - si esce dallo stato attraverso una transizione esterna

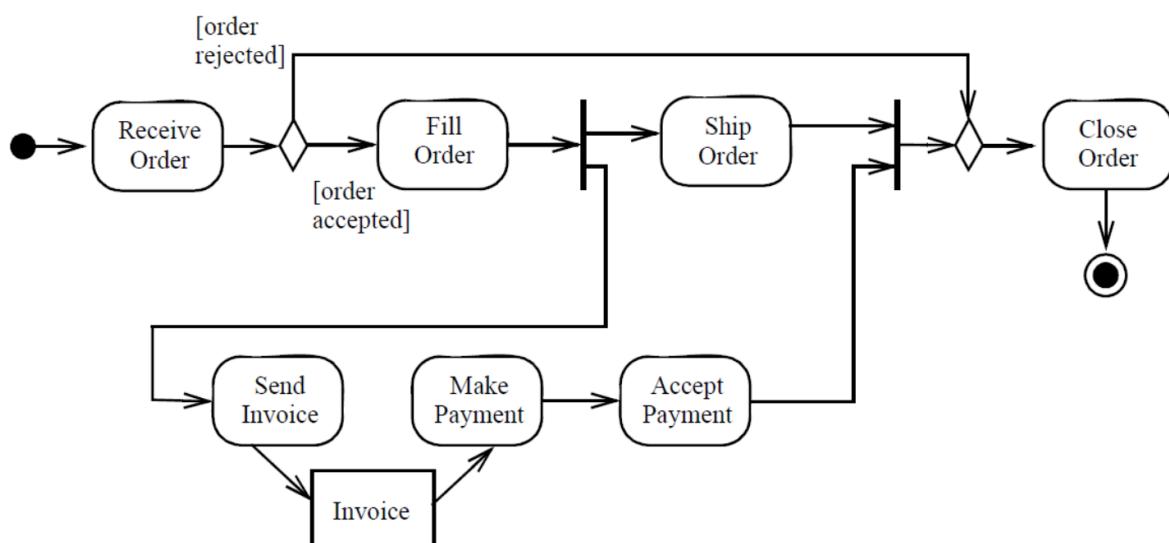


- Il diagramma di stato relativo a una singola classe / entità dovrebbe essere il più semplice possibile

- Le classi / entità con diagrammi di stato complicati hanno diversi problemi:
 - il codice risulta più difficile da scrivere perché le implementazioni dei metodi contengono molti blocchi condizionali
 - il testing è più arduo
 - è molto difficile utilizzare una classe dall'esterno se il comportamento dipende dallo stato in modo complesso
- Se una classe ha un numero troppo elevato di stati è bene considerare se esistono progetti alternativi, come per esempio scomporre la classe in due diverse classi
- Questa però non va presa come regola universale
- La scomposizione di una classe è sempre un'operazione molto delicata e va ponderata molto bene in quanto potrebbe portare a progetti fuorvianti o instabili
- A volte il male minore è proprio avere diagrammi di stato complessi

1.10 Diagramma delle attività

- I diagrammi delle attività descrivono il modo in cui diverse attività sono coordinate e possono essere usati per mostrare l'implementazione di una operazione
- Un diagramma delle attività mostra le attività di un sistema in generale e delle sotto-parti, specialmente quando un sistema ha diversi obiettivi e si desidera modellare le dipendenze tra essi prima di decidere l'ordine in cui svolgere le azioni
- I diagrammi delle attività sono utili anche per descrivere lo svolgimento dei singoli casi d'uso e la loro eventuale dipendenza da altri casi
- Sostanzialmente, modellano un processo
- Organizzano più entità in un insieme di azioni secondo un determinato flusso
- Si usano ad esempio per:
 - modellare il flusso di un caso d'uso (analisi)
 - modellare il funzionamento di un'operazione (progettazione)
 - modellare un algoritmo (progettazione)

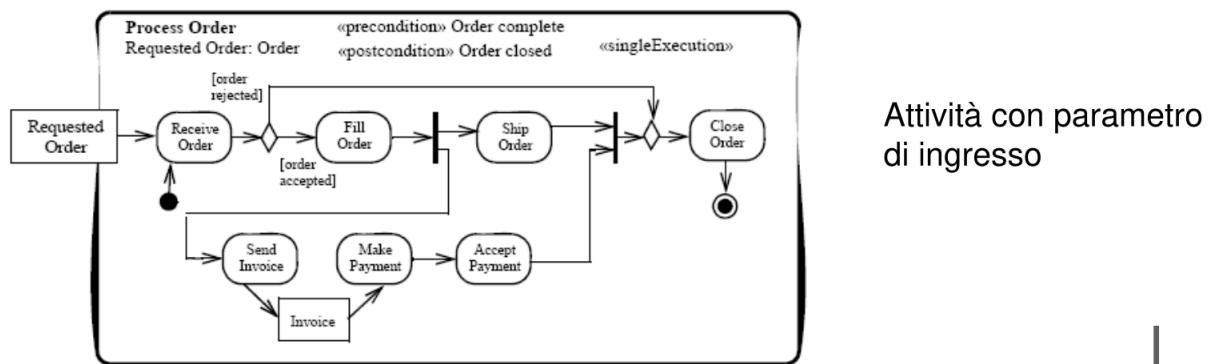
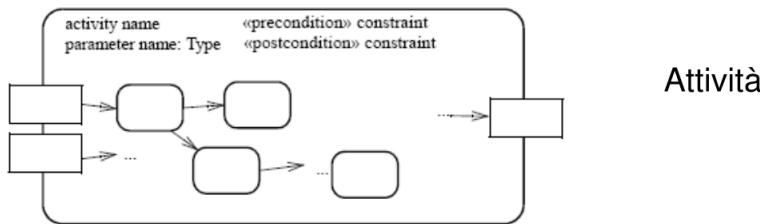


- **Attività:** indica un lavoro che deve essere svolto
 - Specifica un'unità atomica, non interrompibile e istantanea di comportamento
 - Da ogni attività possono uscire uno o più archi, che indicano il percorso da una attività ad un'altra
- **Arco:** è rappresentato come una freccia proprio come una transizione
 - A differenza di una transizione, un arco non può essere etichettato con eventi o azioni

- Un arco può essere etichettato con una condizione di guardia, se l'attività successiva la richiede
- **Oggetto:** rappresenta un oggetto “importante” usato come input/output di azioni
- **Sottoattività:** “nasconde” un diagramma delle attività interno a un’attività
- **Start e End Point:** punti di inizio e fine del diagramma
 - Gli End Point possono anche non essere presenti, oppure essere più di uno
- **Controllo:** nodi che descrivono il flusso delle attività

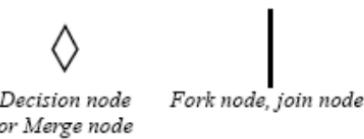
- Per capire la semantica dei diagrammi di attività, bisogna immaginare delle entità, dette **token**, che viaggiano lungo il diagramma
- Il flusso dei token definisce il flusso dell’attività
- I token possono rimanere fermi in un nodo azione/oggetto in attesa che si avveri
 - una condizione su un arco
 - oppure una precondizione o postcondizione su un nodo
- Il movimento di un token è atomico
- Un nodo azione viene eseguito quando sono presenti token su tutti gli archi in entrata, e tutte le precondizioni sono soddisfatte
- Al termine di un’azione, sono generati token su tutti gli archi in uscita

1.10.1.1 Notazione



- **Decision e Merge:** determinano il comportamento condizionale
 - **Decision** ha una singola transizione entrante e più transizioni uscenti in cui solo una di queste sarà prescelta
 - **Merge** ha più transizioni entranti e una sola uscente e serve a terminare il blocco condizionale cominciato con un decision
- **Fork e join:** determinano il comportamento parallelo:
 - quando scatta la transizione entrante, si eseguono in parallelo tutte le transazioni che escono dal fork
 - Con il parallelismo non è specificata la sequenza
 - Le fork possono avere delle guardie

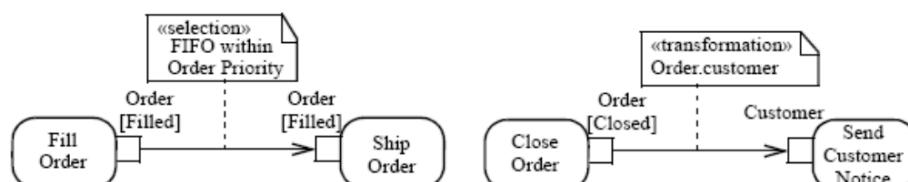
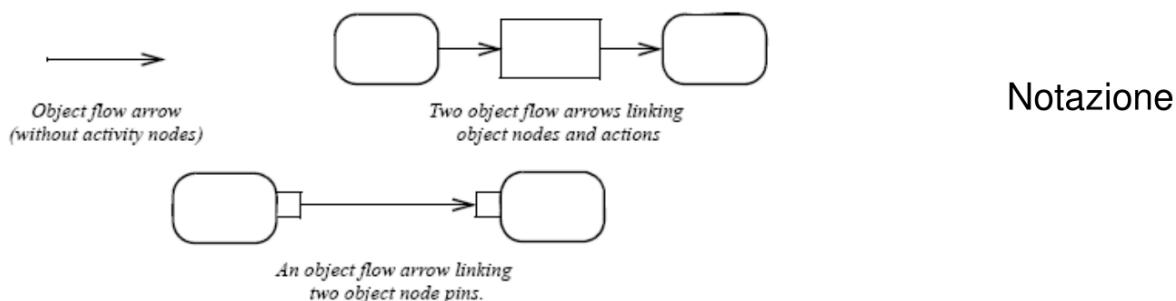
- Per la sincronizzazione delle attività è presente il costrutto di join che ha più transazioni entranti e una sola transazione uscente



{:height 343, :width 434}

1.10.2 Object Flow

- È un arco che ha oggetti o dati che fluiscono su di esso

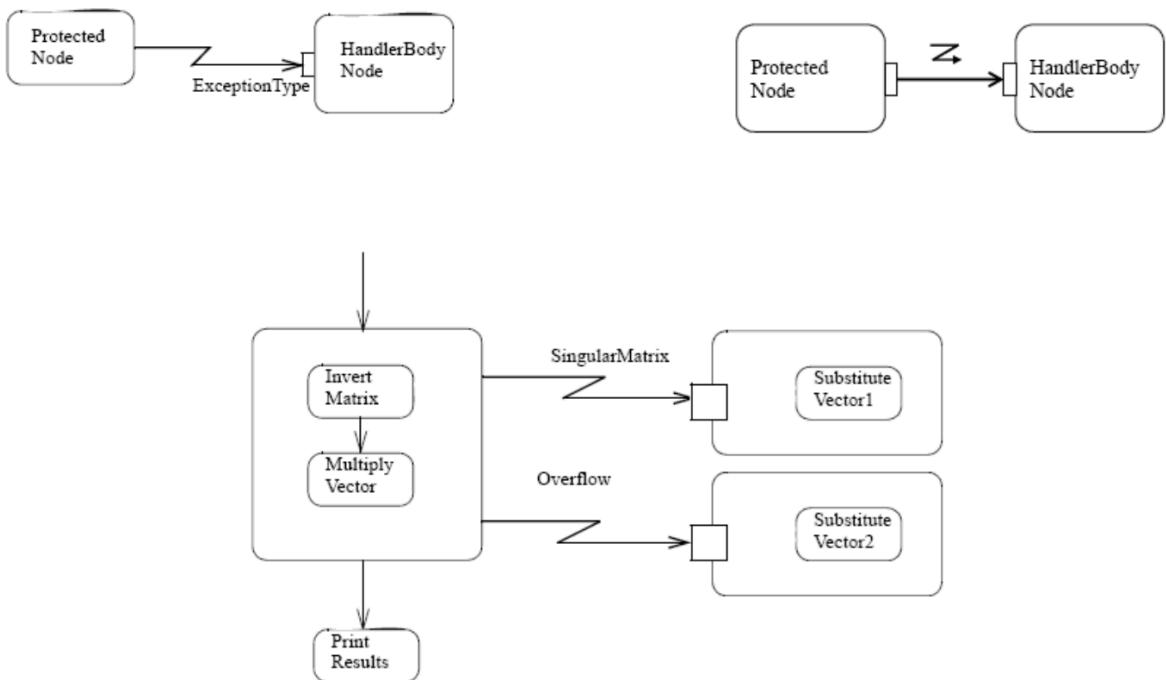


Esempio

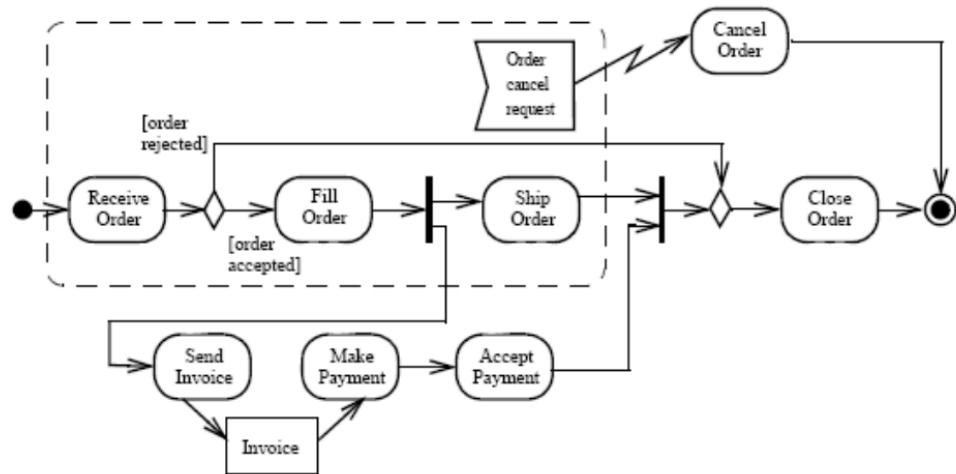
1.10.3 Segnali ed Eventi



1.10.4 Exception Handler



1.10.5 InterruptibleActivityRegion



2 Progetto

2.1 Analisi dei requisiti

2.1.1 Requisiti

- I requisiti di un sistema rappresentano la descrizione
 - dei servizi forniti (le funzionalità)
 - dei vincoli operativi
- Il processo di ricerca, analisi, documentazione e verifica di questi servizi e vincoli è chiamato **Ingegneria dei Requisiti** (RE - Requirements Engineering)
- I requisiti di solito vengono forniti a livelli diversi di descrizione e questo porta a una prima classificazione...
- I Requisiti utente dichiarano:
 - Quali servizi il sistema dovrebbe fornire
 - I vincoli sotto cui deve operare
 - Sono requisiti molto astratti e di alto livello che vengono specificati nella prima fase interlocutoria con il committente
 - Tipicamente sono espressi in linguaggio naturale e corredati da qualche diagramma
- Requisiti di sistema definiscono:
 - Le funzioni, i servizi e i vincoli operativi del sistema in modo dettagliato
 - È una descrizione dettagliata di quello che il sistema dovrebbe fare
 - Il Documento dei Requisiti del Sistema deve essere preciso e definire esattamente cosa deve essere sviluppato
 - Tale documento fa spesso parte del contratto tra committente e azienda sviluppatrice
- ![[image.png]](images/image_1710063362882_0.png)

2.1.1.1 Requisiti di Sistema

- I requisiti di sistema tipicamente vengono divisi in tre diverse tipologie:
 - Requisiti funzionali
 - Requisiti non funzionali
 - Requisiti di dominio

2.1.1.2 Requisiti Funzionali

- Descrivono quello che il sistema dovrebbe fare
- Sono elenchi di servizi che il sistema dovrebbe fornire
- Per ogni servizio dovrebbe essere indicato:
 - come reagire a particolari input
 - come comportarsi in particolari situazioni
 - in alcuni casi specificare cosa il sistema NON dovrebbe fare
- Le specifiche dei requisiti funzionali dovrebbero essere:
 - complete: tutti i servizi definiti
 - coerenti: i requisiti non devono avere definizioni contraddittorie

2.1.1.3 Requisiti Non Funzionali

- Non riguardano direttamente le specifiche funzionali
- Non devono essere necessariamente coerenti
- I principali tipi di requisiti non funzionali:
 - Requisiti del prodotto: specificano o limitano le proprietà complessive del sistema
 - affidabilità, prestazioni, protezione dei dati, disponibilità dei servizi, tempi di risposta, occupazione di spazio, capacità dei dispositivi di I/O, rappresentazione dei dati nelle interfacce, etc.
 - Requisiti Organizzativi: possono vincolare anche il processo di sviluppo adottato
 - politiche e procedure dell'organizzazione cliente e sviluppatrice, specifiche degli standard di qualità da adottare, uso di un particolare CASE tool, linguaggi di implementazione, limiti di budget, requisiti di consegna e milestone, etc.
 - Requisiti esterni: si identificano tutti i requisiti che derivano da fattori non provenienti dal sistema e dal suo processo di sviluppo
 - necessità di interoperabilità con altri sistemi software o hardware prodotti da altre organizzazioni
 - requisiti legislativi che devono essere rispettati affinché il sistema operi nella legalità → legislazioni sulla privacy dei dati
 - requisiti etici perché il sistema possa essere accettato dagli utenti e dal grande pubblico
- ![[image.png]](images/image_1710063695471_0.png)
- Uno dei maggiori problemi di questi requisiti è che possono essere difficili da verificare perché spesso sono espressi in modo vago e sono mescolati ai requisiti funzionali
 - Esempi: facilità d'uso, capacità di ripristino dopo un malfunzionamento
- Spesso contrastano o interagiscono con i requisiti funzionali
 - La protezione e la privacy dei dati contrastano con la facilità d'uso perché richiedono procedure spesso macchinose per l'utente...
 - ...occorre trovare un compromesso tra i requisiti o chiedere al committente quale sia più prioritario

- Vanno studiati ed analizzati con molta cura e precisione e indicati quanto più chiaramente possibile nel documento dei requisiti

2.1.1.4 Requisiti di Dominio

- I requisiti di dominio descrivono i vincoli che esistono nel dominio e che potrebbero non essere condivisi tra tutti coloro che saranno impegnati nello sviluppo del software
- Derivano dal dominio di applicazione del sistema
- Solitamente includono una terminologia propria del dominio del sistema o si riferiscono ai suoi concetti
- Poiché sono requisiti “specialistici” spesso gli ingegneri del software trovano difficile capire come questo tipo di requisiti si rapportino agli altri requisiti del sistema
- Sono requisiti che vanno comunque analizzati con molta cura perché riflettono i fondamenti del dominio dell’applicazione
 - l’analisi deve coinvolgere gli esperti del dominio per chiarire ogni dubbio sulla terminologia

2.1.2 Analisi

- Obiettivo
 - Specificare (cioè definire) le proprietà che il sistema dovrà avere senza descrivere una loro possibile realizzazione
- Risultato: una serie di documenti
 - contenenti la descrizione dettagliata dei requisiti
 - base di partenza per l’analisi del problema
- Per determinare in dettaglio i requisiti del sistema, è necessario
 - interagire il più possibile con l’utente, o con il potenziale utente
 - conoscere il più possibile l’area applicativa

1. Raccolta dei requisiti

Obiettivo: raccogliere tutte le informazioni su cosa il sistema deve fare secondo le intenzioni del cliente

2. Analisi dei requisiti

Obiettivo: definire il comportamento del sistema

3. Analisi del dominio

Obiettivo: definire la porzione del mondo reale, rilevante per il sistema

4. Analisi e gestione dei rischi

Obiettivo: identificare e gestire i possibili rischi che possono fare fallire o intralciare la realizzazione del sistema ![image.png](images/image_1710064987414_0.png)

2.1.2.1 Raccolta dei Requisiti

- Obiettivo: raccogliere tutte le informazioni su cosa il sistema deve fare secondo le intenzioni del cliente (si chiede al cliente cosa voglia che faccia l’applicazione)
- Non prevede passi formali, né ha una notazione specifica, perché dipende moltissimo dal particolare tipo di problema
- Risultato
 - un documento (testuale)
 - scritto dall’analista
 - discusso e approvato dal cliente

- ▶ una versione iniziale del vocabolario o glossario contenente la definizione precisa e non ambigua di tutti i termini e i concetti utilizzati
- Tipologie di persone coinvolte
 - ▶ Analista
 - ▶ Utente
 - ▶ Esperto del dominio (non sempre indispensabile)
- Metodi utilizzati
 - ▶ Interviste, questionari
 - ▶ Studio di documenti che esprimono i requisiti in forma testuale
 - ▶ Osservazione passiva o attiva del processo da modellare
 - ▶ Studio di sistemi software esistenti
 - ▶ Prototipi
- La gestione delle interviste è molto complessa
- I clienti potrebbero
 - ▶ Avere solo una vaga idea dei requisiti
 - ▶ Non essere in grado di esprimere i requisiti in termini comprensibili
 - ▶ Chiedere requisiti non realizzabili o troppo costosi
 - ▶ Fornire requisiti in conflitto
 - ▶ Essere poco disponibili a collaborare

2.1.2.2 Validazione dei Requisiti

- Ogni requisito deve essere validato e negoziato con i clienti prima di essere riportato nel Documento dei Requisiti (se non è completamente validato si rifà l'intervista)
- Attività svolta in parallelo alla raccolta
- **Validità** - il nuovo requisito è inerente il problema da risolvere?
- **Consistenza** - il nuovo requisito è in sovrapposizione e/o in conflitto con altri requisiti?
- **Realizzabilità** - il nuovo requisito è realizzabile con le risorse disponibili (hardware, finanziamenti, ...)?
- **Completezza** - esiste la possibilità che ci siano requisiti rimasti ignorati?

2.1.2.3 Documento dei Requisiti

- Il Documento dei Requisiti deve specificare in modo chiaro e univoco cosa il sistema dovrà fare
- I requisiti dovrebbero essere:
 - ▶ Chiari
 - ▶ Corretti
 - ▶ Completati
 - ▶ Concisi
 - ▶ Non ambigui
 - ▶ Precisi
 - ▶ Consistenti
 - ▶ Tracciabili
 - ▶ Modificabili
 - ▶ Verificabili
- Il Documento dei Requisiti deve contenere la versione iniziale del dizionario dei termini

- Un buon modo per organizzare i requisiti e facilitare la tracciabilità è quello di elencare i requisiti in una tabella
- ! [image.png] (images/image_1710065295316_0.png)
- Se durante il processo di sviluppo ci si riferisce sempre allid del requisito diventa facile collegare le diverse fasi e garantire una tracciabilità requisito-codice

2.1.2.4 Cambiamento dei Requisiti

- È normale che i requisiti subiscano modificazioni ed evolvano nel tempo
- Requisiti esistenti possono essere rimossi o modificati
- Nuovi requisiti possono essere aggiunti in una fase qualsiasi del ciclo di sviluppo
- Tali cambiamenti
 - Sono la norma, non l'eccezione; i sistemi vanno progettati avendo in mente il cambiamento
 - Possono diventare un grosso problema se non opportunamente gestiti
- Più lo sviluppo è avanzato, più il cambiamento è costoso
 - Modificare un requisito appena definito è facile
 - Modificare lo stesso requisito dopo che è stato implementato nel software potrebbe essere molto costoso
- Ogni cambiamento deve essere accuratamente analizzato per valutare
 - La fattibilità tecnica, cioè, lo si può implementare, indipendentemente dal costo?
 - L'impatto sul resto del sistema
 - Il costo
- Consiglio - sviluppare sistemi che
 - Siano il più possibile resistenti ai cambiamenti dei requisiti
 - Inizialmente, eseguano esclusivamente e nel modo migliore i soli compiti richiesti
 - In seguito, siano in grado di sostenere l'aggiunta di nuove funzionalità senza causare "disastri" strutturali e/o comportamentali
- Tenete traccia dei cambiamenti anche nella tabella dei requisiti

2.1.2.5 Analisi del Dominio

- Obiettivo: definire la porzione del mondo reale rilevante per il sistema, quindi una visione astratta del mondo reale
- Principio fondamentale: Astrazione Permette di gestire la complessità intrinseca del mondo reale
 - ignorare gli aspetti che non sono importanti per lo scopo attuale
 - concentrarsi maggiormente su quelli che lo sono
- Risultato: prima versione del vocabolario partendo dai "sostantivi" che si trovano nei requisiti
- L'analisi del dominio può essere effettuata anche considerando un gruppo di sistemi afferenti alla stessa area applicativa
- Esempi di aree applicative:
 - il controllo del traffico aereo
 - la gestione aziendale
 - le operazioni bancarie
 - ...
- In tal caso, è possibile
 - identificare entità e comportamenti comuni a tutti i sistemi

- ▶ realizzare schemi di progettazione e componenti software riutilizzabili nei diversi sistemi

2.1.2.6 Analisi dei Requisiti

- Obiettivo: definire il comportamento del sistema da realizzare
- Risultato: un modello comportamentale (o modello dinamico) che descrive in modo chiaro e conciso le funzionalità del sistema
- L'analisi dei requisiti deve partire dalla raccolta, che è un elenco di requisiti, e definire per bene qual è il comportamento del sistema
- Strategia:
 - ▶ Scomposizione funzionale (mediante analisi top-down) ▶ identificare le singole funzionalità previste dal sistema
 - ▶ Astrazione procedurale ▶ considerare ogni operazione come una singola entità, nonostante tale operazione sia effettivamente realizzata da un insieme di operazioni di più basso livello
- Attenzione: La scomposizione in funzioni è molto volatile a causa del continuo cambiamento dei requisiti funzionali
- Come prima cosa vanno analizzati in modo sistematico tutti i requisiti inseriti nella Tabella dei requisiti
- Bisogna porre particolare attenzione ai sostantivi e ai verbi che compaiono nel testo di specifica dei requisiti
- Dall'analisi dei sostantivi è possibile formalizzare la conoscenza sul dominio applicativo → costruzione di un primo modello del dominio
- Dall'analisi dei verbi è possibili individuare linsieme delle azioni che il sistema dovrà compiere → modello dei casi duso
- Aggiornare costantemente la Tabella dei Requisiti → dall'analisi nascono sempre nuovi requisiti

2.1.2.7 Vocabolario

- Nella modellazione del dominio è di fondamentale importanza usare solo la terminologia di quello specifico dominio
- Il vocabolario è una lista dei termini usati nella specifica dei requisiti a cui viene data una definizione precisa e non ambigua
- È uno dei fattori chiave per migliorare la comunicazione tra i diversi attori del processo di sviluppo, in particolare tra analisti e progettisti
- Ciascuna entità del dominio che si evince dai requisiti può essere espressa come classe UML e messa in relazione logica con le altre entità andando a creare il primo modello del dominio

2.1.2.8 Analisi e gestione dei rischi

- Ci si riferisce generalmente ai rischi di tipo legale
- Analisi sistematica e completa di tutti i possibili rischi che possono fare fallire o intralciare la realizzazione del sistema in una qualsiasi fase del processo di sviluppo
- Ogni rischio presenta due caratteristiche:
 - ▶ Probabilità che avvenga non esistono rischi con una probabilità del 100% (sarebbero vincoli al progetto)
 - ▶ Costo se il rischio si realizza, ne seguono effetti indesiderati e/o perdite
- Rischi relativi ai requisiti
 - ▶ I requisiti sono perfettamente noti?

- Il rischio maggiore è quello di costruire un sistema che non soddisfa le esigenze del cliente
- Rischi relativi alle risorse umane
 - È possibile contare sulle persone e sull'esperienza necessarie per lo sviluppo del progetto?
- Rischi relativi alla protezione e privacy dei dati
 - Di che tipo sono i dati che dobbiamo trattare?
 - Quali sono i possibili attacchi informatici a cui il sistema può essere soggetto?
- Rischi tecnologici
 - Si sta scegliendo la tecnologia corretta?
 - Si è in grado di aggregare correttamente i vari componenti del progetto (ad es., GUI, DB, ...)?
 - Quali saranno i possibili futuri sviluppi della tecnologia?
- Rischi politici
 - Ci sono delle forze politiche (anche in senso lato) in grado di intralciare lo sviluppo del progetto?
- Strategia reattiva o “alla Indiana Jones”
 - “Niente paura, troverò una soluzione”
- Strategia preventiva
 - Si mette in moto molto prima che inizi il lavoro tecnico
 - Si individuano i rischi potenziali, se ne valutano le probabilità e gli effetti e si stabilisce un ordine di importanza
 - Si predisponde un piano che permetta di reagire in modo controllato ed efficace
 - Più grande è un rischio, dove per grande possiamo intendere il prodotto di probabilità per costo
 - Maggiore sarà l'attenzione che bisognerà dedicargli

2.1.3 Sicurezza e Privacy

2.1.3.1 Nuova normativa

2.1.3.1.1 General Data Protection Regulation

- Dal 25/5/2018 sostituisce la Data Protection Directive
- Obbligo di aderenza (compliance) di un prodotto software che tratti dati personali ai principi della GDPR
 - privacy by design & by default(misure tecniche e organizzative)
 - minimalità, proporzionalità
 - anonimizzazione, pseudonimizzazione
 - trasferimento dati fuori dalla EU (occhio al cloud!)
 - adeguatezza delle misure di sicurezza
- Non è qualcosa che si possa “aggiungere dopo”, a sistema già progettato: va considerato fin dall'inizio
 - ma non è (solo) un vincolo: è un'opportunità per creare valore

Pseudonimizzazione: processo di trattamento dei dati personali in modo tale che i dati non possano più essere attribuiti a un interessato specifico senza l'utilizzo di informazioni aggiuntive, sempre che tali informazioni aggiuntive siano conservate separatamente e soggette a misure tecniche e organizzative intese a garantire la non attribuzione a una persona identificata o identificabile.

2.1.3.1.2 Principi

I dati personali devono essere:

- trattati in modo lecito, equo e trasparente nei confronti dell'interessato (“[liceità, equità e trasparenza](#)”)
- raccolti per finalità determinate, esplicite e legittime, e successivamente trattati in modo non incompatibile con tali finalità
- adeguati, pertinenti e limitati a quanto necessario rispetto alle finalità per le quali sono trattati (“[minimizzazione dei dati](#)”)
- esatti e, se necessario, aggiornati
 - devono essere prese tutte le misure ragionevoli per cancellare o rettificare tempestivamente i dati inesatti rispetto alle finalità per le quali sono trattati (“[esattezza](#)”); quindi bisogna dare all'utente la possibilità di aggiornarli
- conservati in una forma che consenta l'identificazione degli interessati per un arco di tempo non superiore al conseguimento delle finalità per le quali sono trattati
 - i dati personali possono essere conservati per periodi più lunghi a condizione che siano trattati esclusivamente per finalità di archiviazione nel pubblico interesse o per finalità di ricerca scientifica e storica o per finalità statistiche, conformemente all'articolo 83
- trattati in modo da garantire un'adeguata sicurezza dei dati personali, compresa la protezione, mediante misure tecniche e organizzative adeguate, da trattamenti non autorizzati o illeciti e dalla perdita, dalla distruzione o dal danno accidentali (“[integrità e riservatezza](#)”); quindi bisogna garantire che non ci siano trattamenti illeciti dei dati degli utenti, e che non vengano persi

2.1.3.1.3 Articolo 25

Protezione dei dati fin dalla progettazione e protezione di default

1. Tenuto conto dello stato dell'arte e dei costi di attuazione, nonché della natura, del campo di applicazione, del contesto e delle finalità del trattamento, come anche dei rischi di varia probabilità e gravità per i diritti e le libertà delle persone fisiche costituiti dal trattamento, [sia al momento di determinare i mezzi del trattamento sia all'atto del trattamento stesso](#) il responsabile del trattamento mette in atto [misure tecniche e organizzative adeguate](#), quali la pseudonimizzazione, volte ad attuare i principi di protezione dei dati, quali la minimizzazione, in modo efficace e a integrare nel trattamento le necessarie garanzie al fine di soddisfare i requisiti del presente regolamento e tutelare i diritti degli interessati
2. Il responsabile del trattamento mette in atto misure tecniche e organizzative adeguate per garantire che siano trattati, [di default](#), solo i dati personali necessari per ogni specifica finalità del trattamento; ciò vale per la quantità dei dati raccolti, l'estensione del trattamento, il periodo di conservazione e l'accessibilità. [In particolare dette misure garantiscono che, di default, non siano resi accessibili dati personali a un numero indefinito di persone fisiche senza l'intervento della persona fisica](#); cioè, normalmente, nessuno può far vedere quei dati, tranne la persona stessa

2.1.3.1.4 Articolo 32:

Sicurezza del Trattamento

1. Tenuto conto dello stato dell'arte e dei costi di attuazione, nonché della natura, del campo di applicazione, del contesto e delle finalità del trattamento, come anche del rischio di varia probabilità e gravità per i diritti e le libertà delle persone fisiche, il responsabile del trattamento e l'incaricato del trattamento mettono in atto misure tecniche e organizzative adeguate per garantire un **livello di sicurezza adeguato al rischio**, che comprendono tra l'altro:
 - la **pseudonimizzazione** e la **cifratura dei dati personali**
 - la capacità di assicurare la **continua riservatezza**, integrità, disponibilità e resilienza dei sistemi e dei servizi che trattano i dati personali;
 - la capacità di **ripristinare tempestivamente la disponibilità** e l'accesso dei dati in caso di incidente fisico o tecnico;
 - una procedura per provare, verificare e valutare regolarmente l'efficacia delle misure tecniche e organizzative al fine di garantire la sicurezza del trattamento; cioè bisogna non solo mettere in campo queste misure, ma anche testarle regolarmente
2. Nel valutare l'adeguato livello di sicurezza, si tiene conto in special modo [dei rischi presentati] da trattamenti di dati derivanti in particolare dalla distruzione, dalla perdita, dalla modifica, dalla divulgazione non autorizzata o dall'accesso, in modo accidentale o illegale, a dati personali trasmessi, memorizzati o comunque trattati
3. L'adesione a un **codice di condotta** approvato di cui all'articolo 40 o a un **meccanismo di certificazione** approvato di cui all'articolo 42 può essere utilizzata come elemento per dimostrare la conformità ai requisiti di cui al paragrafo 1 del presente articolo

Quindi per dimostrare di aver seguito tutti i passi descritti nell'articolo ci si avvale di una certificazione o di un codice di condotta.

2.1.3.1.5 Trasferimento Dati a Paesi Terzi

- Articolo 45:
 - Il trasferimento di dati personali verso un paese terzo o un'organizzazione internazionale è ammesso se la Commissione ha deciso che il paese terzo, o un territorio o uno o più settori specifici all'interno del paese terzo, o l'organizzazione internazionale in questione garantiscano un livello di protezione adeguato. In tal caso il trasferimento non necessita di autorizzazioni specifiche.
- Articolo 46:
 - In mancanza di una decisione ai sensi dell'articolo 45, il responsabile del trattamento o l'incaricato del trattamento può trasferire dati personali verso un paese terzo o un'organizzazione internazionale solo se ha offerto garanzie adeguate e a condizione che siano disponibili diritti azionabili degli interessati e mezzi di ricorso effettivi per gli interessati.

2.1.3.2 Concetti di Base

2.1.3.2.1 Sicurezza Informatica

Salvaguardia dei sistemi informatici da potenziali rischi e/o violazioni dei dati

- Obiettivo dell'attacco = contenuto informativo

- Sicurezza informatica = preoccuparsi di
 - impedire l'accesso ad utenti non autorizzati
 - regolamentare l'accesso ai diversi soggetti...
 - ... che potrebbero avere autorizzazioni diverse (relative solo a certe operazioni e non altre)
- per evitare che i dati appartenenti al sistema informatico vengano copiati, modificati o cancellati

Cosa Proteggere:

- L'informazione...
 - Riservatezza: solo determinati utenti possono avere accesso alle informazioni
 - Integrità ed Autenticità: nessuno deve modificare impropriamente quelle informazioni
 - Disponibilità: le informazioni sono disponibili nel momento c'è la necessità di accedervi
- trattata per mezzo di calcolatori e reti
 - Accesso controllato
 - Tre aspetti: **identificazione, autenticazione, autorizzazione**
 - Funzionamento affidabile; per il concetto di affidabilità riguardare gli argomenti precedenti

2.1.3.2.2 Violazioni

- Le violazioni possono essere molteplici:
 - tentativi non autorizzati di accesso a zone riservate
 - furto di identità digitale o di file riservati
 - utilizzo di risorse che l'utente non dovrebbe potere utilizzare
 - ecc.
- La sicurezza informatica si occupa anche di prevenire eventuali **Denial of Service** (DoS)
 - sono attacchi sferrati al sistema con l'obiettivo di rendere inutilizzabili alcune risorse onde danneggiare gli utenti

2.1.3.2.3 Fattori Influenti

- Nella scelta delle misure di sicurezza **incidono diverse caratteristiche** dell'informazione e del contesto:
 - Dinamicità
 - Dimensione e tipo di accesso
 - Tempo di vita
 - Costo di generazione
 - Costo in caso di violazione (di riservatezza, di integrità, di disponibilità)
 - Valore percepito (dall'utente e dal detentore) e tipologia di attaccante

La Catena degli Anelli

La forza di un sistema è pari alla forza dell'anello più debole che lo compone. Non bisogna tralasciare nulla, nemmeno il più piccolo dettaglio

2.1.3.2.4 Metodi di Protezione

- Legali: ci sono leggi che regolano i casi in cui un sistema informatico viene violato, attraverso sanzioni ad esempio
- Organizzativi
- Tecnologici: metodi che ci interessano di più
 - Fisici: un oggetto fisico che viene utilizzato per proteggere l'accesso ai dati, ad esempio un tesserino
 - Crittografici: cifrare i dati in modo che siano difficilmente interpretabili da un attaccante
 - Biometrici: caratteristiche fisiche misurabili che identificano univocamente un individuo

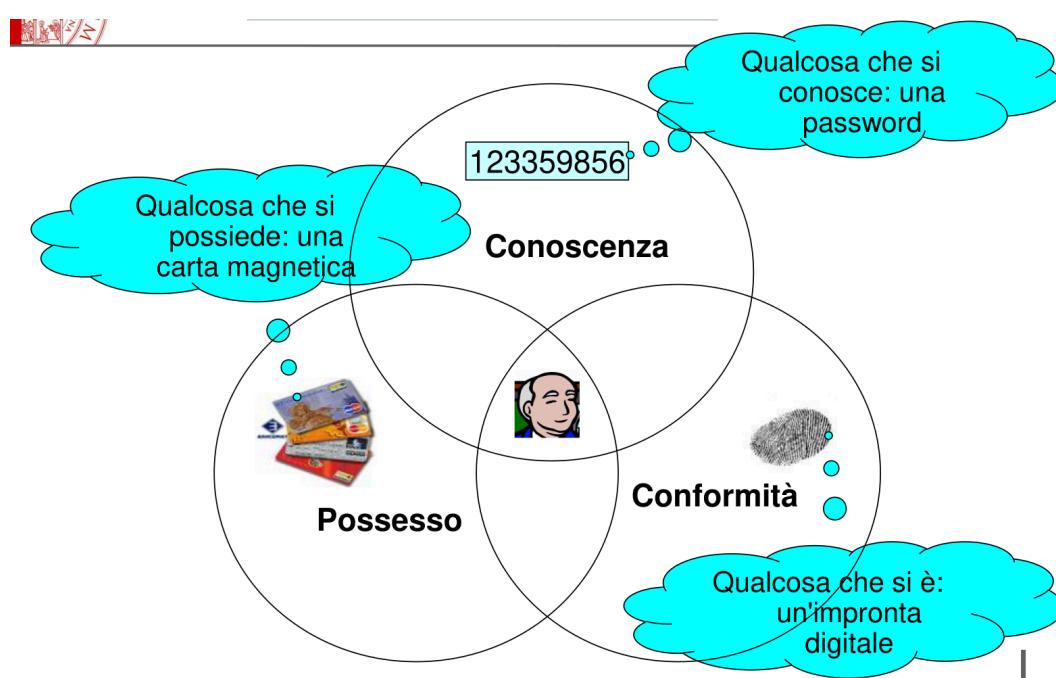
Attuiamo questi metodi per:

- Prevenzione: evitare che un attaccante entri nel sistema
- Rilevazione: scopriamo eventuali attacchi o violazioni
- Contenimento: se c'è una violazione dei sistemi si cerca di tutti i modi di contenere eventuali danni
- Ripristino: ripristino del sistema all'istante precedente al danno

2.1.3.2.5 Protezione Fisica

- Essenziale: la vulnerabilità fisica non sia la più facilmente attaccabile
- Efficace per i sistemi
 - criteri che esistono da ben prima dei problemi di sicurezza informatica
- Efficace per i dati
 - purché si conosca il comportamento dei sistemi che li trattano (percorsi accessibili, copie temporanee in memoria e su disco, ...)
 - costo di generazione
 - purché si prevedano metodi aggiuntivi per contenere gli effetti delle violazioni fisiche dei sistemi (es. furto)

2.1.3.2.6 Autenticazione Forte



L'autenticazione forte è garantita esistono le seguenti condizioni

- Possesso
- Conoscenza

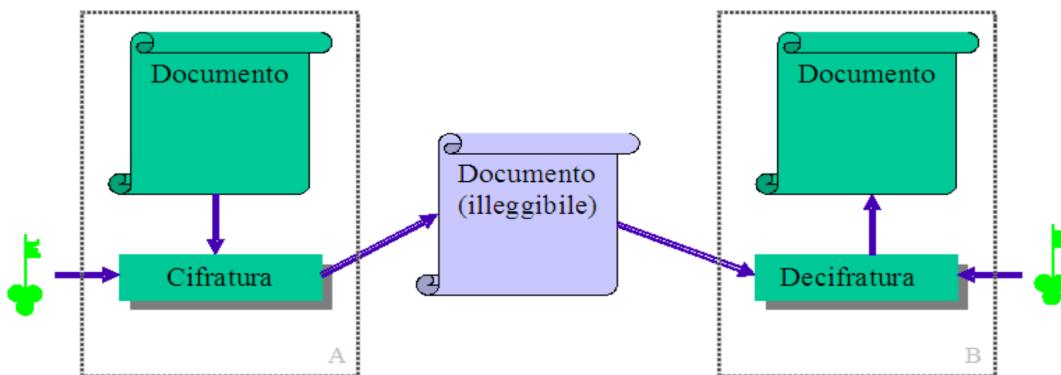
- Conformità

Ovviamente garantire tutte e tre le condizioni ha un costo.

2.1.3.3 Crittografia

- Crittografia simmetrica
 - garantire riservatezza
 - non identifica né autentica
 - **funzionamento:** esiste una sola chiave che viene utilizzata per cifrare; ad esempio prendo un dato, lo cifro, e ottengo un dato che non è immediatamente comprensibile; la stessa chiave è usata per decifrare.
- Crittografia asimmetrica
 - garantisce riservatezza
 - obiettivo: identificazione e quindi autenticazione e paternità
 - **funzionamento:** le chiavi sono due; c'è una che è in esclusivo possesso del cfratore e una che invece è pubblica, quindi chiave privata e chiave pubblica
- Infrastrutture per la certificazione della chiave pubblica
 - Terze parti fidate che possano certificare l'autenticità di una chiave pubblica

2.1.3.3.1 Crittografia Simmetrica



- Classica e moderna, implementata con dispositivi segreti, algoritmi segreti o chiavi segrete
- Tipicamente tecniche derivanti dalla teoria dell'informazione (confusione e diffusione)
- Una singola chiave cifra e decifra

2.1.3.3.2 Crittografia Asimmetrica

- Moderna (ufficialmente 1976)
- Basata sulla teoria della complessità computazionale
- Due chiavi correlate ma non (facilmente) calcolabili l'una dall'altra
 - La **chiave privata** è strettamente personale e quindi *identifica* il possessore
 - L'uso di una determinata chiave privata può essere verificato da chiunque per mezzo della corrispondente **chiave pubblica**

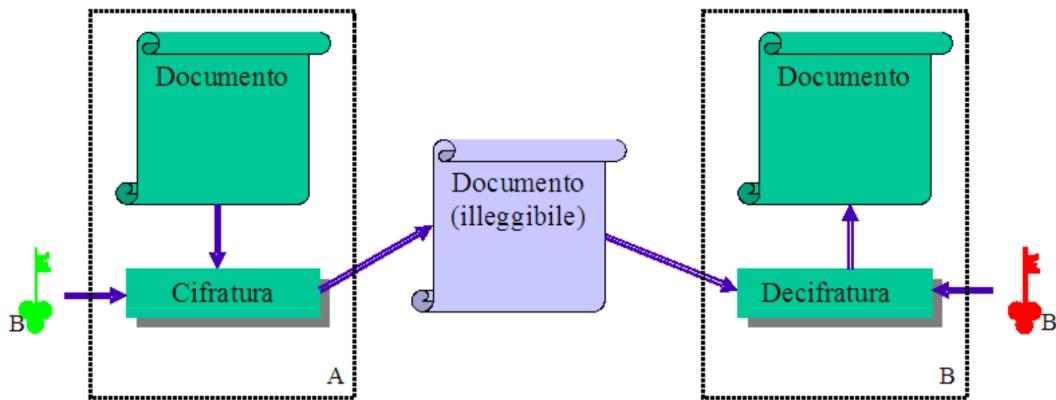


Figure 64: Procedimento di cifratura

Se io voglio che il mio documento sia letto solamente da chi deve poterlo leggere, io cifro quel documento con la chiave pubblica, quindi se io cifro con la chiave privata è decifrabile solo dalla chiave pubblica, se io cifro con la chiave pubblica è decifrabile solo dalla chiave privata

Firma Digitale:

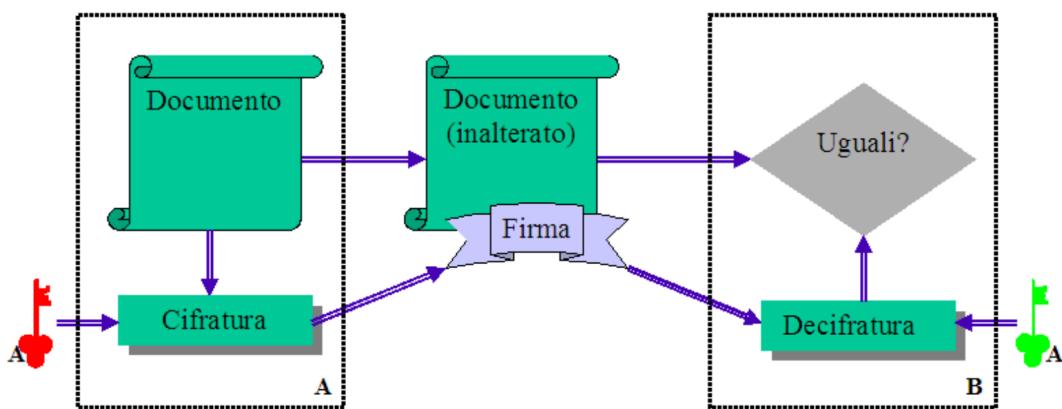


Figure 65: Procedimento di firma

Viceversa, firma, autenticità: se io cifro il documento con la mia chiave pubblica genero un documento che è leggibile, ma che di fianco ha la mia firma, a questo punto chiunque sia in possesso della chiave pubblica può cifrare o decifrare la firma, e vedere se i documenti sono uguali, cioè controllo che la chiave che è stata usata per cifrare, per produrre la firma, corrisponde alla chiave pubblica che ho in mano io.

2.1.3.3.3 Confronto tra i due tipo di crittografia

	Simmetrica	Asimmetrica
Autenticità		
Integrità		
Riservatezza		
Efficienza		
Robustezza		
Chiavi		
Lunghezza		
Numero per ogni utente		
Protezione		

La cifratura asimmetrica purtroppo è poco efficiente perché le procedure di cifratura e decifratura evidentemente sono complicate. Però, per quanto riguarda le chiavi simmetriche, bisogna averne un numero importante, perché se ho una sola chiave che cifra e decifra tutto, se capita in mano a un malintenzionato ha accesso a tutto.

2.1.3.3.4 Biometria

- Componente cardine per la terna di fattori per l'**autenticazione forte**
 - qualcosa che sei, qualcosa che hai, qualcosa che sai
- Problemi non ancora risolti:
 - Ostinatamente usata per l'*identificazione*
 - lunghe operazioni di confronto
 - cattivo bilanciamento tra falsi positivi e falsi negativi
 - Meglio per l'*autenticazione*
 - un solo confronto, minore probabilità di errori
 - Prestazioni più sfumate rispetto ad altre tecniche
 - difficile tuning tra falsi positivi e falsi negativi
 - Impossibilità di sostituzione in caso di compromissione

2.1.3.3.5 Sicurezza delle Password

- Aspetto da sempre fondamentale per:
 - impedire l'accesso a utenti non autorizzati
 - nascondere e/o vincolare l'accesso a documenti
- Diverse categorie:
 - Deboli
 - Semplici
 - Intelligenti
 - Strong

2.1.3.4 Analizzare e Progettare la Sicurezza

2.1.3.4.1 Sistema Sicuro

- La definizione di una politica di sicurezza deve tenere conto di vincoli tecnici, logistici, amministrativi, politici ed economici, imposti dalla struttura organizzativa in cui il sistema opera

- Per questo serve introdurre la sicurezza sin dalle prime fasi di analisi dei requisiti di un nuovo sistema
 - le vigenti leggi, le politiche e i vincoli aziendali sono la base di partenza per la definizione di un **piano per la sicurezza**
- Un'applicazione o un servizio possono consistere di uno o più componenti funzionali allocati localmente o distribuiti sulla rete
- La sicurezza viene vista come un **processo complesso**, come una catena di caratteristiche
 - dalla computer system security, network security, application-level security sino alle problematiche di protezione dei dati sensibili
- La sfida maggiore lanciata ai progettisti è quella di **progettare applicazioni sicure e di qualità** che tengano conto in modo strutturato di tutti gli aspetti della sicurezza sin dalle prime fasi di analisi del sistema

2.1.3.4.2 Sistemi Critici

- I **sistemi critici** sono sistemi tecnici o socio-tecnici da cui dipendono persone o aziende
- Se questi sistemi non forniscono i loro servizi come ci si aspetta possono verificarsi seri problemi e importanti perdite
- Ci sono tre tipi principali di sistemi critici:
 - **Sistemi safety-critical:** i fallimenti possono provocare incidenti, perdita di vite umane o seri danni ambientali
 - **Sistemi mission-critical:** i malfunzionamenti possono causare il fallimento di alcune attività e obiettivi diretti
 - **Sistemi business-critical:** i fallimenti possono portare a costi molto alti per le aziende che li usano
- La proprietà più importante di un sistema critico è la sua **fidatezza**
- Sistema fidato = **disponibilità + affidabilità + sicurezza e protezione**
- Ci sono diverse ragioni per le quali la fidatezza è importante:
 - I sistemi non affidabili, non sicuri e non protetti sono rifiutati dagli utenti
 - I costi di un fallimento del sistema potrebbero essere enormi
 - I sistemi inaffidabili possono causare perdita di informazioni
- Componenti del sistema che possono causare fallimenti (in ordine dal più affidabile al meno affidabile):
 - Hardware: può fallire a causa di errori nella progettazione, di un guasto a un componente o perché i componenti hanno terminato la loro vita naturale
 - Software: può fallire a causa di errori nelle sue specifiche, nella sua progettazione o nella sua implementazione
 - Operatori umani: possono sbagliare a interagire con il sistema
- Con l'aumentare dell'affidabilità di software e hardware gli errori umani sono diventati la più probabile causa di difetto di un sistema. Una persona può non usarlo come dovrebbe, come abbiamo pensato che dovesse utilizzarlo o che volontariamente usa il sistema in maniera inopportuna per evitare la fidatezza del

sistema e quindi per distruggere l'affidabilità, distruggere la sicurezza, distruggere la protezione e distruggere la disponibilità.

- La sicurezza e la protezione dei sistemi critici sono diventate sempre più importanti con l'aumentare delle connessioni di rete
- Da una parte le connessioni di rete espongono il sistema ad attacchi da parte di malintenzionati
- Dall'altra parte la rete fa in modo che i dettagli delle vulnerabilità siano facilmente divulgati e facilita la distribuzione di patch
- Esempi di attacchi sono:
 - virus
 - usi non autorizzati dei servizi
 - modifiche non autorizzate al sistema e ai suoi dati
 - **Exploit:** metodo che sfrutta un bug o una vulnerabilità, per l'acquisizione di privilegi
 - **Buffer Overflow:** fornire al programma più dati di quanto esso si aspetti di ricevere, in modo che una parte di questi vadano scritti in zone di memoria dove sono, o dovrebbero essere, altri dati o lo stack del programma stesso
 - **Shell code:** sequenza di caratteri che rappresenta un codice binario in grado di lanciare una shell, può essere utilizzato per acquisire un accesso alla linea di comando
 - **Sniffing:** attività di intercettazione passiva dei dati che transitano in una rete
 - **Cracking:** modifica di un software per rimuovere la protezione dalla copia, oppure per ottenere accesso a un'area riservata
 - **Spoofing:** tecnica con la quale si simula un indirizzo IP privato da una rete pubblica facendo credere agli host che l'IP della macchina server da contattare sia il suo
 - **Trojan:** programma che contiene funzionalità maliziose; la vittima è indotta a eseguire il programma poiché questo viene spesso inserito nei videogiochi pirati
 - **Denial of Service:** il sistema viene forzatamente messo in uno stato in cui i suoi servizi non sono disponibili, influenzando così la disponibilità del sistema

2.1.3.5 Introduzione alla Security Engineering

2.1.3.5.1 Security Engineering

Security Engineering is concerned with how to develop and maintain systems that can resist malicious attacks intended to damage a computer-based system or its data



Da: Software Engineering 8 – I.Sommerville - Addison Wesley – Cap. 30

- L'ingegneria della sicurezza è parte del più vasto campo della sicurezza informatica
- Nell'ingegnerizzazione di un sistema software non si può prescindere dalla consapevolezza delle minacce che il sistema dovrà affrontare e dei modi in cui tali minacce possono essere neutralizzate
- Quando si considerano le problematiche di sicurezza nell'ingegnerizzazione di un sistema vanno presi in considerazione due aspetti diversi:

- la sicurezza dell'applicazione
- la sicurezza dell'infrastruttura su cui il sistema è costruito



2.1.3.5.2 Applicazione e Infrastruttura

- La sicurezza di una applicazione è un problema di ingegnerizzazione del software dove gli ingegneri devono garantire che il **sistema sia progettato per resistere agli attacchi**
- La sicurezza dell'infrastruttura è invece un problema manageriale nel quale gli amministratori dovrebbero garantire che **l'infrastruttura sia configurata per resistere agli attacchi**
- Gli amministratori dei sistemi devono:
 - inizializzare l'infrastruttura in modo tale che tutti i servizi di sicurezza siano disponibili
- monitorare e riparare eventuali falle di sicurezza che emergono durante l'uso del software

2.1.3.5.3 Gestione della Sicurezza

- Gestione degli utenti e dei permessi:
 - inserimento e rimozione di utenti dal sistema
 - autenticazione degli utenti
 - creazione di appropriati permessi per gli utenti
- Deployment e mantenimento del sistema:
 - installazione e configurazione dei software e del middleware
 - aggiornamento periodico del software con tutte le patch disponibili
- Controllo degli attacchi, rilevazione e ripristino
 - controllo del sistema per accessi non autorizzati
 - identificazione e messa in opera di strategie contro gli attacchi
 - backup per ripristinare il normale utilizzo dopo un attacco

2.1.3.6 Glossario e Minacce

2.1.3.6.1 Glossario della Sicurezza

- **Bene** (Asset): una risorsa del sistema che deve essere protetta
- **Esposizione** (Exposure): possibile perdita o danneggiamento come risultato di un attacco andato a buon fine
 - Potrebbe essere una perdita o un danneggiamento di dati o una perdita di tempo nel ripristino del sistema dopo l'attacco

- **Vulnerabilità** (Vulnerability): una debolezza nel sistema software che potrebbe essere sfruttata per causare una perdita o un danno
- **Attacco** (Attack): sfruttamento di una vulnerabilità del sistema allo scopo di esporre un bene
- **Minaccia** (Threat): circostanza che ha le potenzialità per causare perdite e danni
- **Controllo** (Control): una misura protettiva che riduce una vulnerabilità del sistema

2.1.3.6.2 Tipi di Minacce

- **Minacce alla riservatezza del sistema o dei suoi dati**: le informazioni possono essere svelate a persone o programmi non autorizzati
- **Minacce all'integrità del sistema o dei suoi dati**: i dati o il software possono essere danneggiati o corrotti
- **Minacce alla disponibilità del sistema o dei suoi dati**: può essere negato l'accesso agli utenti autorizzati al software o ai dati
- Queste minacce sono interdipendenti
 - Se un attacco rende il sistema non disponibile, la modifica sulle informazioni potrebbe non avvenire, rendendo così il sistema non integro

2.1.3.6.3 Tipi di Controllo

- **Controlli per garantire che gli attacchi non abbiano successo**: la strategia è quella di progettare il sistema in modo da evitare i problemi di sicurezza
 - i sistemi militari sensibili non sono connessi alla rete pubblica
 - crittografia
- **Controlli per identificare e respingere attacchi**: la strategia è quella di monitorare le operazioni del sistema e identificare pattern di attività atipici, nel caso agire di conseguenza (spegnere parti del sistema, restringere l'accesso agli utenti, ..)
- **Controlli per il ripristino**
 - backup, replicazione, polizze assicurative

Esempio

- Un sistema informativo ospedaliero mantiene le informazioni personali sui pazienti e sui loro trattamenti
- Il sistema deve essere accessibile da differenti ospedali e cliniche attraverso un'interfaccia web
- Lo staff ospedaliero deve utilizzare una specifica coppia <username, password> per autenticarsi, dove lo username è il nome del dipendente
- Il sistema richiede password che siano lunghe almeno 8 caratteri, ma consente ogni password senza ulteriori verifiche

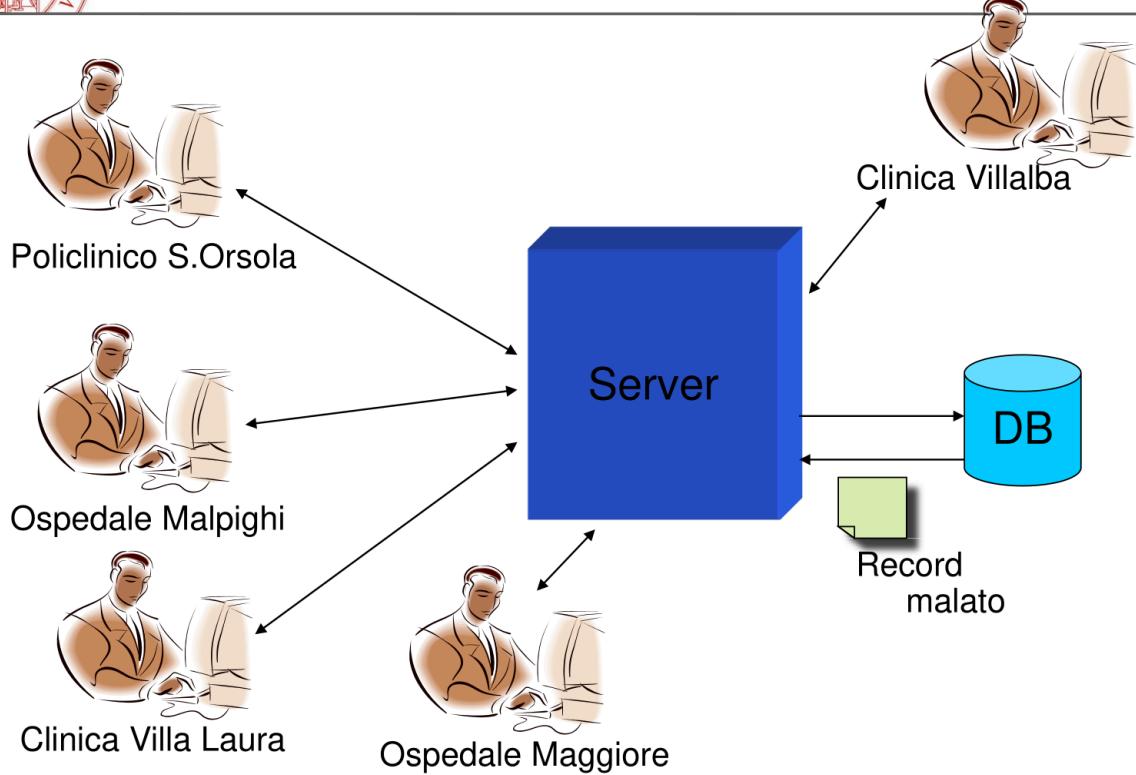


Figure 69: Schema logico

Termine	Descrizione
Beni	I record dei pazienti che ricevono o hanno ricevuto trattamenti sanitari; il database; il sistema informativo
Esposizione	Possibile perdita di futuri pazienti che non si fidano della clinica. Perdita finanziaria e perdita d'immagine
Vulnerabilità	Un sistema di password debole rende facile agli utenti la memorizzazione di password banali; lo username è uguale al nome del dipendente
Attacchi	Furto di identità di un utente autorizzato e successiva violazione e sottrazione di dati; denial of service
Minacce	Possibilità di indovinare le password di utenti autorizzati; arrivo contemporaneo di un numero elevato di richieste
Controllo	Sistema di controllo delle password che obblighi gli utenti a utilizzare password di tipo strong; replicazione del servizio su più server

2.1.3.7 Analisi del Rischio

- L'analisi del rischio si occupa di
 - valutare le possibili perdite che un attacco può causare ai beni di un sistema
 - bilanciare queste perdite con i costi richiesti per la protezione dei beni stessi

costo protezione << costo della perdita

- L'analisi del rischio è una problematica più manageriale che tecnica
- Il ruolo degli ingegneri della sicurezza è quindi quello di fornire una [guida tecnica e giuridica](#) sui problemi di sicurezza del sistema

- Sarà poi compito dei manager decidere se accettare i costi della sicurezza o i rischi che derivano dalla mancanza di procedure di sicurezza
- L'analisi del rischio inizia dalla valutazione delle politiche di sicurezza di organizzazione che spiegano cosa dovrebbe e cosa non dovrebbe essere consentito fare
- Le politiche di sicurezza propongono le condizioni che dovrebbero sempre essere mantenute dal sistema di sicurezza, quindi aiutano ad identificare le minacce che potrebbero sorgere
- La valutazione del rischio è un processo in più fasi:
 - **valutazione preliminare del rischio**: determina i requisiti di sicurezza dell'intero sistema
- **ciclo di vita della valutazione del rischio**: avviene contestualmente e segue il ciclo di vita dello sviluppo del software; valutare i rischi ogni volta che viene cambiato qualcosa

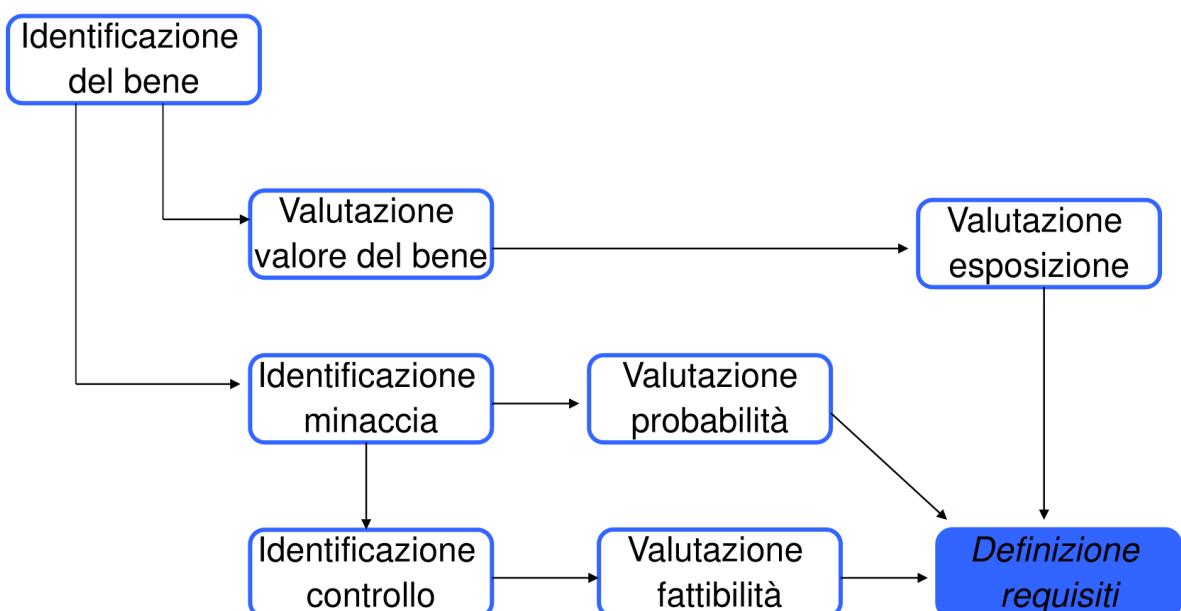


Figure 72: Valutazione Preliminare del Rischio

2.1.3.8 Identificazione del bene

2.1.3.8.1 Analisi del Sistema Informatico

- Durante questa fase si può stabilire la seguente agenda delle attività:
 - Analisi delle risorse fisiche
 - Analisi delle risorse logiche
 - Analisi delle dipendenze fra risorse

2.1.3.8.2 Analisi delle Risorse Fisiche

- In questa attività, il sistema informatico viene visto come insiemi di dispositivi che, per funzionare, hanno bisogno di spazio, alimentazione, condizioni ambientali adeguate, protezioni da furti e danni materiali
- In particolare occorre:
 - individuare sistematicamente tutte le risorse fisiche
 - ispezionare e valutare tutti i locali che ospiteranno le risorse fisiche
 - verificare la cablatura dei locali

2.1.3.8.3 Analisi delle Risorse Logiche

- Il sistema viene visto come insieme di informazioni, flussi e processi
- In particolare occorre:
 - **Classificare le informazioni** in base al valore che hanno per l'organizzazione, il grado di riservatezza e il contesto di afferenza
 - **Classificare i servizi** offerti dal sistema informatico affinché non presentino effetti collaterali pericolosi per la sicurezza del sistema

2.1.3.8.4 Analisi delle Dipendenze tra Risorse

- Per ciascuna risorsa (fisica o logica) occorre individuare di quali altre risorse essa ha bisogno per funzionare correttamente
- Questa analisi tende ad evidenziare le risorse potenzialmente critiche, ovvero quelle da cui dipende il funzionamento di un elevato numero di altre risorse
- I risultati di questa analisi sono usati anche nella fase di valutazione del rischio
 - in particolare, sono di supporto allo studio della propagazione dei malfunzionamenti a seguito dell'occorrenza di eventi indesiderati

2.1.3.9 Identificazione delle minacce

- In questa fase si cerca di definire quello che **non deve poter accadere** nel sistema
- Si parte dal considerare come evento indesiderato qualsiasi accesso che non sia esplicitamente permesso
- A tal fine è possibile in generale distinguere tra
 - attacchi intenzionali
 - eventi accidentali

2.1.3.9.1 Attacchi Intenzionali

- Gli attacchi vengono caratterizzati in funzione della risorsa (sia fisica che logica) che viene attaccata e delle possibili tecniche usate per l'attacco
- Le tecniche di attacco possono essere classificate in funzione del livello al quale operano
- Si distingue tra **tecniche a livello fisico** e **a livello logico**
 - Gli attacchi a livello fisico sono principalmente tesi a sottrarre o danneggiare le risorse critiche
 - Si tratta di
 - Furto = un attacco alla disponibilità e alla riservatezza
 - Danneggiamento = un attacco alla disponibilità e alla integrità

2.1.3.9.2 Attacchi a Livello Logico

- Gli attacchi a livello logico sono principalmente tesi a sottrarre informazione o degradare l'operatività del sistema
- Dal punto di vista dei risultati che è indirizzato a conseguire un attacco logico può essere classificato come:
- **Intercettazione e deduzione** (attacco alla riservatezza): sniffing, spoofing, emulatori...
- **Intrusione** (attacco all'integrità e alla riservatezza): IP-spoofing, backdoor...

- Disturbo (attacco alla disponibilità): virus, worm, denial of service...

2.1.3.9.3 Eventi Accidentali

- Una possibile casistica degli eventi accidentali che accadono più di frequente:
- **a livello fisico:**
 - guasti ai dispositivi che compongono il sistema
 - guasti di dispositivi di supporto (es. condizionatori)
- **a livello logico:**
 - perdita di password o chiave hardware
 - cancellazione di file
 - corruzione del software di sistema (ad esempio, a seguito dell'installazione di estensioni incompatibili)

2.1.3.9.4 Valutazione dell'Esposizione

- A ogni minaccia occorre associare un rischio così da indirizzare l'attività di individuazione delle contromisure verso le aree più critiche
- Per **rischio** s'intende una combinazione della probabilità che un evento accada con il danno che l'evento può arrecare al sistema
- Nel valutare il danno si tiene conto
 - delle dipendenze tra le risorse
 - dell'eventuale propagazione del malfunzionamento

2.1.3.9.5 Valutazione delle Probabilità: Attacchi Intenzionali

- La probabilità di occorrenza di attacchi intenzionali dipende principalmente dalla **facilità** di attuazione e dai **vantaggi** che potrebbe trarne l'intruso
 - Il danno si misura come **grado di perdita** dei tre requisiti fondamentali (riservatezza, integrità, disponibilità)
- MA l'attaccante applicherà sempre tutte le tecniche di cui dispone, su tutte le risorse attaccabili → necessità di valutare anche il rischio

di un **attacco composto**

- un insieme di attacchi elementari concepiti con un medesimo obiettivo e condotti in sequenza

2.1.3.9.6 Individuazione del Controllo

- Occorre scegliere il controllo da adottare per neutralizzare gli attacchi individuati:
 - Valutazione del rapporto costo/efficacia
 - Analisi di standard e modelli di riferimento
 - Controllo di carattere organizzativo
 - Controllo di carattere tecnico

2.1.3.9.7 Valutazione del Rapporto Costo/Efficienza

- Valuta il **grado di adeguatezza** di un controllo
- Mira ad evitare che i controlli presentino un costo ingiustificato rispetto al rischio dal quale proteggono
- **Efficienza del controllo** definita come funzione del rischio rispetto agli eventi indesiderati che neutralizza

- Il costo di un controllo deve essere calcolato senza dimenticare i [costi nascosti](#)

2.1.3.9.8 Costi Nascosti

- Occorre tenere presenti le limitazioni che i controlli impongono e le operazioni di controllo che introducono nel flusso di lavoro del sistema informatico e dell'organizzazione
- Le principali voci di costo sono le seguenti:
 - costo di messa in opera del controllo
 - peggioramento dell'ergonomia dell'interfaccia utente
 - decadimento delle prestazioni del sistema nell'erogazione dei servizi
 - aumento della burocrazia

2.1.3.9.9 Controlli di Carattere Organizzativo

- [Condizione essenziale](#) affinché la tecnologia a protezione del sistema informatico risulti efficace è che venga [utilizzata nel modo corretto](#) da personale pienamente [consapevole](#)
- Devono quindi essere definiti con precisione [ruoli e responsabilità](#) nella gestione sicura di tale sistema
- Per ciascun ruolo, dall'amministratore al semplice utente, devono essere definite [norme comportamentali e procedure precise](#) da rispettare

2.1.3.9.10 Controlli di Carattere Tecnico

- Controlli di base
 - a livello del sistema operativo e dei servizi di rete
- Controlli specifichi del particolare sistema
 - si attestano normalmente a livello applicativo
- Controlli tecnici più frequenti
 - configurazione sicura del sistema operativo di server e postazioni di lavoro (contromisura di base)
 - confinamento logico delle applicazioni server su server dedicati
- Etichettatura delle informazioni, allo scopo di avere un controllo più fine dei diritti di accesso
 - Moduli software di cifratura integrati con le applicazioni
 - Apparecchiature di telecomunicazione in grado di cifrare il traffico dati in modo trasparente alle applicazioni
 - Firewall e server proxy in corrispondenza di eventuali collegamenti con reti TCP/IP
 - Chiavi hardware e/o dispositivi di riconoscimento degli utenti basati su rilevamenti biofisici
- Integrazione dei Controlli
- Un insieme di controlli non deve presentarsi come una “collezione di espedienti” non correlati tra loro
- È importante integrare i vari controlli

in una politica di sicurezza organica

- Operare una selezione dei controlli adottando un sottoinsieme di costo minimo che rispetti alcuni vincoli:

- completezza delle contromisure
- omogeneità delle contromisure
- ridondanza controllata delle contromisure
- effettiva attuabilità delle contromisure

Vincoli del Sottoinsieme

- Completezza:

il sottoinsieme deve fare fronte a tutti gli eventi indesiderati

- Omogeneità:

le contromisure devono essere compatibili e integrabili tra loro

- Ridondanza controllata:

la ridondanza delle contromisure ha un costo e deve essere rilevata e vagliata accuratamente

- Effettiva attuabilità:

l'insieme delle contromisure deve rispettare tutti i vincoli imposti dall'organizzazione nella quale andrà ad operare

Esempio

- Torniamo all'esempio del sistema per la gestione di dati ospedalieri

- Quali sono i beni del sistema?
- il sistema informativo
- il database dei pazienti
- i record di ogni paziente

Quali sono le minacce?

- furto identità dell'amministratore
- furto identità di utente autorizzato
- DoS

Esempio

Valutazione del Bene

Bene Valore Esposizione

Sistema

Informativo

Alto. Supporto a tutte le consultazioni cliniche

Alta. Perdita finanziaria; costi ripristino sistema; danni a pazienti se cure non date

Database pazienti

Alto. Supporto a tutte le consultazioni cliniche Critico dal punto di vista della sicurezza

Alta. Perdita finanziaria; costi ripristino sistema; danni a pazienti se cure non date

Record paziente

Normalmente basso. Potrebbe essere alto per pazienti particolari

Perdita diretta bassa, ma possibile perdita di reputazione della clinica

Esempio

Analisi Minacce e Controlli

Minaccia Probabilità Controllo Fattibilità

Furto identità

Amministratore

Bassa Accesso solo da postazioni specifiche fisicamente sicure

Basso costo implementativo ma attenzione alla distribuzione delle chiavi

Furto identità utente

Alta Meccanismi biometrici Molto costoso e non accettato Log di tutte le operazioni

Semplice, trasparente e supporta il ripristino

DoS Media Progettazione adeguata, controllo e limitazione degli accessi

Basso costo. Impossibile prevedere e impedire questo tipo di attacco

Esempio

Requisiti di Sicurezza

- Alcuni dei requisiti ricavati dalla valutazione preliminare dei rischi
- le informazioni relative ai pazienti devono essere scaricate all'inizio della sessione clinica dal database e memorizzate in un'area sicura sul client
- le informazioni relative ai pazienti non devono essere mantenute sul client dopo la chiusura della sessione clinica
- deve essere mantenuto un log su una macchina diversa dal server
- che memorizzi tutti i cambiamenti effettuati sul database

Bene Valore Esposizione Sistema Informativo Alto. Supporto a tutta la gestione del villaggio turistico

Alta. Perdita finanziaria e di immagine Informazioni relative agli Ospiti

Medio. Dati generali degli ospiti del villaggio turistico

Media. Perdita di immagine se vengono divulgati dati degli Ospiti

Informazioni relative alle GuestCard

Alto. L'elenco degli acquisti è associato alle GuestCard

Molto Alta. Perdita finanziaria nel caso gli Ospiticontestino acquisti ingiustamente addebitati. Perdita di immagine Informazioni relative alle vendite

Alto. Sulla base dei movimenti nei Punti di Vendita, la Catena Punti Vendita gestisce i magazzini e i dati fiscali

Alta. Perdita finanziaria se i dati delle vendite e delle forniture non coincidono.
Perdita di immagine se il servizio che vuole essere acquistato per qualche motivo non è presente Informazioni relative al personale

Alto. Ci sono tutte le informazioni relative al personale, comprese le credenziali di accesso

Alta. Perdita finanziaria, se vengono rubate le credenziali del personale si possono perpetuare svariate frodi. Perdita di immagine

Villaggio Turistico

Valutazione del Bene Villaggio Turistico

Analisi Minacce e Controlli Minaccia Probabilità Controllo Fattibilità Furto credenziali Operatore

Alta. La username è fissata e facile da identificare

Accesso da macchine sicure

Basso costo di realizzazione ma attenzione se le macchine vengono lasciate incustodite Log delle Operazioni Basso costo implementativo

Furto credenziali personale dei punti di vendita

Alta. La username è fissata e facile da identificare

Accesso da macchine sicure

Basso costo di realizzazione ma attenzione se le macchine vengono lasciate incustodite Log delle Operazioni Basso costo implementativo

Intercettazione comunicazioni

Alta. Il sistema è distribuito e client/server avvengono tantissime interazioni tra i diversi client e il server.

Cifratura delle comunicazioni

Il costo dipende dal tipo di cifratura scelto. Se simmetrica basso, se asimmetrica più alto dovuto alla necessità di rilascio di coppie di chiavi da un ente di certificazione
Log di tutte le operazioni

Basso costo implementativo

DoS Bassa Progettazione adeguata, controllo e limitazione degli accessi

Basso costo. Impossibile prevenire un DoS

Ciclo di vita della valutazione

del rischio

Ciclo di Vita della Valutazione del Rischio

- È necessaria la conoscenza dell'architettura del sistema
- e dell'organizzazione dei dati
- La piattaforma e il middleware sono già stati scelti,
- così come la strategia di sviluppo del sistema
- Questo significa che si hanno molti più dettagli
- riguardo a che cosa è necessario proteggere
- e sulle possibili vulnerabilità del sistema
- Le vulnerabilità possono essere “ereditate”
- dalle scelte di progettazione ma non solo
- La valutazione del rischio dovrebbe essere parte
- di tutto il ciclo di vita del software: dall'ingegnerizzazione
- dei requisiti al deployment del sistema

Ciclo di Vita della Valutazione del Rischio

- Il processo seguito è simile a quello della valutazione
- preliminare dei rischi, con l'aggiunta di attività riguardanti
- l'identificazione e la valutazione delle vulnerabilità
- La valutazione delle vulnerabilità identifica i beni
- che hanno più probabilità di essere colpiti
- da tali vulnerabilità
- Vengono messe in relazione le vulnerabilità
- con i possibili attacchi al sistema
- Il risultato della valutazione del rischio è
- un insieme di decisioni ingegneristiche
- che influenzano la progettazione o l'implementazione
- del sistema o limitano il modo in cui esso è usato

Esempio

- Si supponga che il provider dei servizi ospedalieri decida di acquistare un prodotto commerciale per la gestione dei dati dei pazienti
- Questo sistema deve essere configurato per ogni tipo di clinica in cui è utilizzato
 - Scelte progettuali del sistema acquistato:
 - autenticazione solo con username e password
 - architettura client-server: il client accede attraverso un'interfaccia web standard
 - l'informazione è presentata agli utenti
 - attraverso una web form editabile,
 - è quindi possibile modificare le informazioni

Esempio: Vulnerabilità

Autenticazione

Login/password

Tecnologia Vulnerabilità

Password banali

Utente rivela password

Architettura

client-server

Denial of Service

Informazioni nella cache

Uso web form editabili

Log dettagliati non possibili

Stessi permessi per gli utenti

Bachi nel browser

Esempio

- Valutate le vulnerabilità del sistema adottato

si deve decidere quali mosse attuare al fine di limitare i rischi associati

- Introduzione di nuovi requisiti di sicurezza
- Introduzione di un meccanismo di verifica delle password
- l'accesso al sistema deve essere permesso
- solo ai client approvati e registrati dall'amministratore
- tutti i client devono avere un solo browser installato
- e approvato dall'amministratore

Villaggio Turistico

Vulnerabilità

Tecnologia Vulnerabilità

Autenticazione username/password

- Password banali
- Utente rivela volontariamente password
- Utente rivela password a seguito di un attacco di
- Ingegneria Sociale

Cifratura comunicazioni Le vulnerabilità dipendono dal tipo di cifratura. Cifratura Simmetrica:

- Tempo di vita della chiave. Più informazioni cifro
- con la stessa chiave più materiale offre per l'analisi
- del testo ad un attaccante
- Lunghezza della chiave
- Memorizzazione della chiave
- Cifratura Asimmetrica:
- Memorizzazione chiave privata

Architettura Client/Server DoS

- Man in the Middle
- Sniffing delle comunicazioni

Security Use Case e Misuse Case Security Use Case e Misuse Case

- I misuse case si concentrano sulle interazioni tra l'applicazione e gli attaccanti che cercano di violarla
- La condizione di successo di un misuse case è l'attacco andato a buon fine
- Questo li rende particolarmente adatti per analizzare le minacce, ma non molto utili per la determinazione dei requisiti di sicurezza
- È invece compito dei security use case specificare i requisiti tramite i quali l'applicazione dovrebbe essere in grado di proteggersi dalle minacce

Security Use Cases -Donald G. Firesmith-JOT Vol. 2, No. 3, May-June 2003

Security Use Case e Misuse Case

MisuseCase Security Use Case

Uso Analizzare e specificare le minacce

Analizzare e specificare i requisiti di sicurezza

Criteri di successo Successo attaccanti Successo applicazione

Prodotto da Security Team Security Team

Usato da Security Team RequirementsTeam

Attori Esterni Attaccanti, Utenti Utenti

Guidato da Analisi vulnerabilità dei beni e analisi minacce

Misusecase

Security Use Case e Misuse Case Esempio

INTRODUCTION

VOL. 2 , NO. 3 JOURNAL OF OBJECT TECHNOLOGY 55

The following table summarizes the primary differences between misuse cases and security use cases.

Misuse Cases Security Use Cases Usage Analyze[^] and[^] specify[^] security[^] threats.

Analyze and specify security requirements Success Criteria Misuser[^] Succeeds[^] Application[^] Succeeds[^] Produced By Security[^] Team[^] Security[^] Team[^] Used By Security[^] Team[^] Requirements[^] Team[^] External Actors Misuser,[^] User[^] User[^] Driven By Asset[^] Vulnerability[^] Analysis[^] Threat Analysis

Misuse Cases

To further illustrate the differences between normal use cases, security use cases, and associated misuse cases, consider Figure 3. The traditional use cases for an automated teller machine might include Deposit Funds, Withdraw Funds, Transfer Funds, and Query Balance, all of which are specializations of a general Manage Accounts use

case. To securely manage one's accounts, one can specify security use cases to control access (identification, authentication, and authorization), ensure privacy (of data and communications), ensure integrity (of data and communications), and ensure nonrepudiation of transactions. The resulting four security use cases specify requirements that protect the ATM and its users from three security threats involving attacks by either crackers or thieves.

Customer Manage Accounts

Deposit Funds

Withdraw Funds

Transfer Funds

Query Balance

Control Access (Security)

Ensure Privacy (Security)

Ensure Integrity (Security)

Ensure Nonrepudiation (Security)

Spoof User (Misuse)

Invade Privacy (Misuse)

Perpetrate Fraud (Misuse)

Cracker

Thief

Misuse Case Misuser

Security Use Case

Fig. 3 : Example Security Use Cases and Misuse Cases Security Use Cases -Donald G. Firesmith-JOT Vol. 2, No. 3, May-June 2003 Esempio: Scenario

Security Use Cases -Donald G. Firesmith-JOT Vol. 2, No. 3, May-June 2003 Villaggio Turistico

Punto Vendita

CatenaPuntiVendita

ElencoVendite

Registrazione

NuovoMovimento

Login

ChiusuraCredito

GestioneOspite

ElencoAcquisti FineSettimana<>

FineVacanza<>

Operatore

OspitePagante

<>

<>

<>

AperturaCredito <>

<><>

<>

Gestione Villaggio Turistico

Garantire Protezione (security) Controllo Accesso (security)

(misuse)Frode SniffingInformazioni(misuse) FurtoCredenziali(misuse)

Truffatore Hacker

Villaggio Turistico

Titolo ControlloAccesso Descrizione Gli accessi al sistema devono essere controllati

Misuse case SniffingInformazioni,FurtoCredenziali Relazioni Precondizioni

L'Attaccante ha immezzato per scoprire la password del user name degli operatori e del personale dei punti vendita

Postcondizioni

Il Sistema blocca momentaneamente l'accesso all'utente in notifica un tentativo di accesso fraudolento
Scenario principale

Sistema Attaccante Dopo aver scoperto qualche username tenta di accedere al sistema inserendo password con un attacco condizionario

Controlla le credenziali immesse e blocca l'accesso nel caso tali

credenziali risultino errate dopo un certo numero di tentativi. Scenario di attacco avvenuto con successo

Sistema Attaccante Attacco condizionato riuscito

Il Sistema controlla le credenziali immesse e consente l'accesso al sistema Naviga tra le maschere del sistema e cerca di capire più informazioni possibili

Il Sistema scrive nel log tutte le operazioni seguite dall'utente Il Sistema controlla periodicamente il log alla ricerca di pattern di accesso atipici e se rileva una notifica un accesso fraudolento

Security Use Case: Linee Guida

- I casi d'uso non dovrebbero mai specificare meccanismi

di sicurezza

- Le decisioni relative ai meccanismi devono essere lasciate alla progettazione
- Requisiti attentamente differenziati

dalle informazioni secondarie

- interazioni del sistema, azioni del sistema e post-condizioni
- sono i soli requisiti

- Evitare di specificare vincoli progettuali non necessari
 - Documentare esplicitamente i percorsi individuali attraverso i casi d'uso al fine di specificare i reali requisiti di sicurezza
 - Basare i security use case su differenti tipi di requisiti di sicurezza
- fornisce una naturale organizzazione dei casi d'uso Security Use Case: Linee Guida
- Documentare le minacce alla sicurezza che giustificano i percorsi individuali attraverso i casi d'uso
 - Distinguere chiaramente tra interazioni degli utenti e degli attaccanti
 - Distinguere chiaramente tra le interazioni che sono visibili esternamente e le azioni nascoste del sistema
 - Documentare sia le precondizioni che le post-condizioni che catturano l'essenza dei percorsi individuali SPECIFICA DEI REQUISITI DI SICUREZZA

Requisiti di Sicurezza

- Non è sempre possibile specificare i requisiti associati alla sicurezza in modo quantitativo
- Quasi sempre questa tipologia dei requisiti è espressa nella forma “non deve”
- definisce comportamenti inaccettabili per il sistema
- non definisce funzionalità richieste al sistema
- L'approccio convenzionale della specifica dei requisiti è basato sul contesto, sui beni da proteggere e sul loro valore per l'organizzazione

Categorie Requisiti di Sicurezza

- Requisiti di identificazione
- specificano se un sistema deve identificare gli utenti
- prima di interagire con loro
- Requisiti di autenticazione
- specificano come identificare gli utenti
- Requisiti di autorizzazione
- specificano i privilegi e i permessi di accesso
- degli utenti identificati
- Requisiti di immunità
- specificano come il sistema deve proteggersi da virus, worm
- e minacce simili

Categorie Requisiti di Sicurezza

- Requisiti di integrità
- specificano come evitare la corruzione dei dati

- Requisiti di scoperta delle intrusioni
- specificano quali meccanismi utilizzare per scoprire gli attacchi al Sistema
- Requisiti di non-ripudiazione
- specificano che una parte interessata in una transazione non può negare il proprio coinvolgimento
- Requisiti di riservatezza
- specificano come deve essere mantenuta la riservatezza delle informazioni

Categorie Requisiti di Sicurezza

- Requisiti di controllo della protezione
- specificano come può essere controllato e verificato l'uso del sistema
- Requisiti di protezione della manutenzione del sistema
- specificano come una applicazione può evitare modifiche autorizzate da un accidentale annullamento dei meccanismi di protezione

Villaggio Turistico

Requisiti di Sicurezza

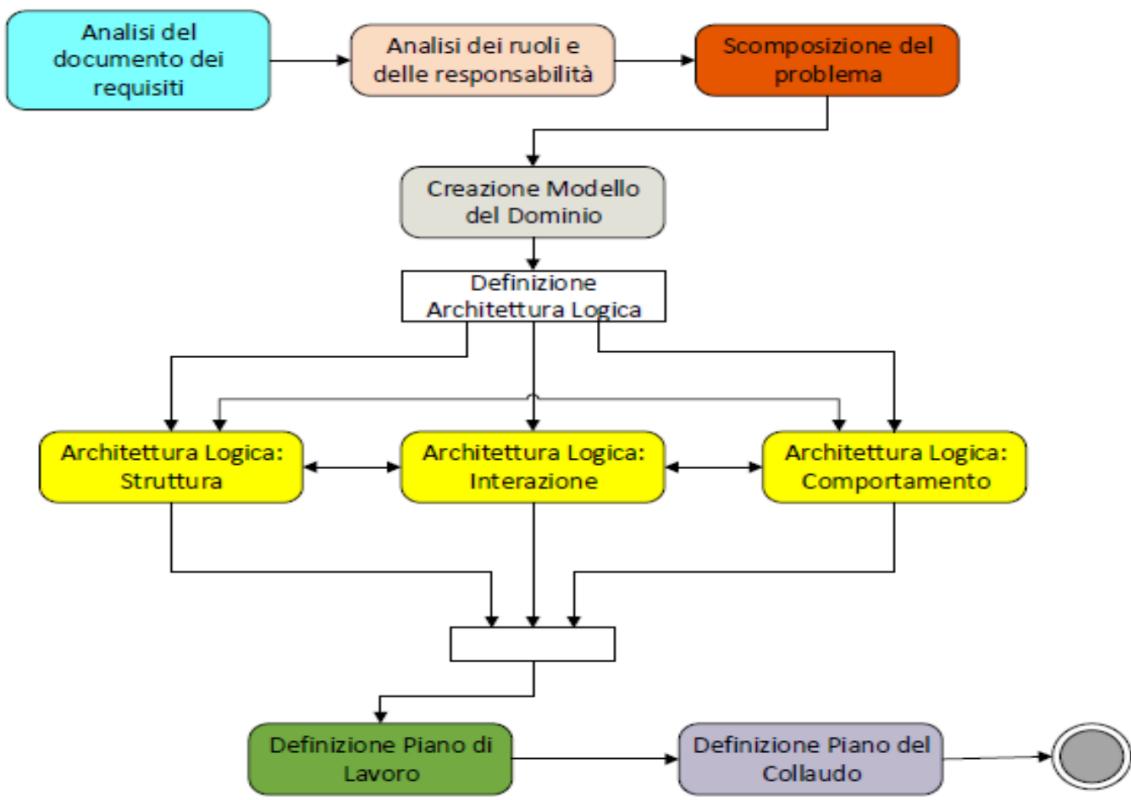
- Creazione di un log per tracciare tutte le azioni che avvengono sul sistema
- i messaggi scambiati tra le parti del sistema
- - che vanno protetti in un qualche modo per evitare che un accesso fraudolento al sistema di log possa rivelare dati riservati
- Adottare meccanismi di analisi del log per identificare pattern di accesso atipici
- identificare discrepanze tra i messaggi spediti e ricevuti
- Individuare una corretta politica di controllo degli accessi
- I dati memorizzati e scambiati nel sistema

devono essere protetti

2.2 Analisi del problema

2.2.1 Introduzione

- **Obiettivo:** esprimere fatti il più possibile “oggettivi” sul **problema** focalizzando l’attenzione su **sottosistemi**, **ruoli** e **responsabilità** insiti negli scenari prospettati durante l’analisi dei requisiti senza descrivere la sua possibile soluzione
- **Risultato:**
 - Architettura Logica
 - Piano di Lavoro
 - Piano del Collaudo (in che modo collauderemo il sistema)



2.2.2 Passi dell'Analisi del Problema

1. Analisi del Documento dei Requisiti

Obiettivo: concentrarsi sull'analisi delle funzionalità e dei rischi evidenziati nel documento dei requisiti

2. Analisi dei Ruoli e delle Responsabilità

Obiettivo: analizzare bene i ruoli emersi nei casi d'uso e porre particolare attenzione nell'attribuzione delle responsabilità a tali ruoli tenendo conto dell'analisi del rischio

3. Scomposizione del problema

Obiettivo: se possibile suddividere il problema in sotto-problemi più piccoli

4. Creazione del Modello del Dominio

Obiettivo: partendo dal vocabolario si costruisce il diagramma delle classi del dominio; il modello di dominio è il modello delle entità che sono importanti nel dominio applicativo e definisce le relazioni tra tali entità

5. Architettura Logica: Struttura

Obiettivo: creazione dei diagrammi strutturali (package e classi) dell'Architettura Logica

6. Architettura Logica: Interazione

Obiettivo: creazione dei diagrammi di interazione (diagrammi di sequenza) dell'Architettura Logica; specificano le funzionalità

7. Architettura Logica: Comportamento

Obiettivo: creazione dei diagrammi di comportamento (stato e attività) dell'Architettura Logica, se è opportuno che ci siano

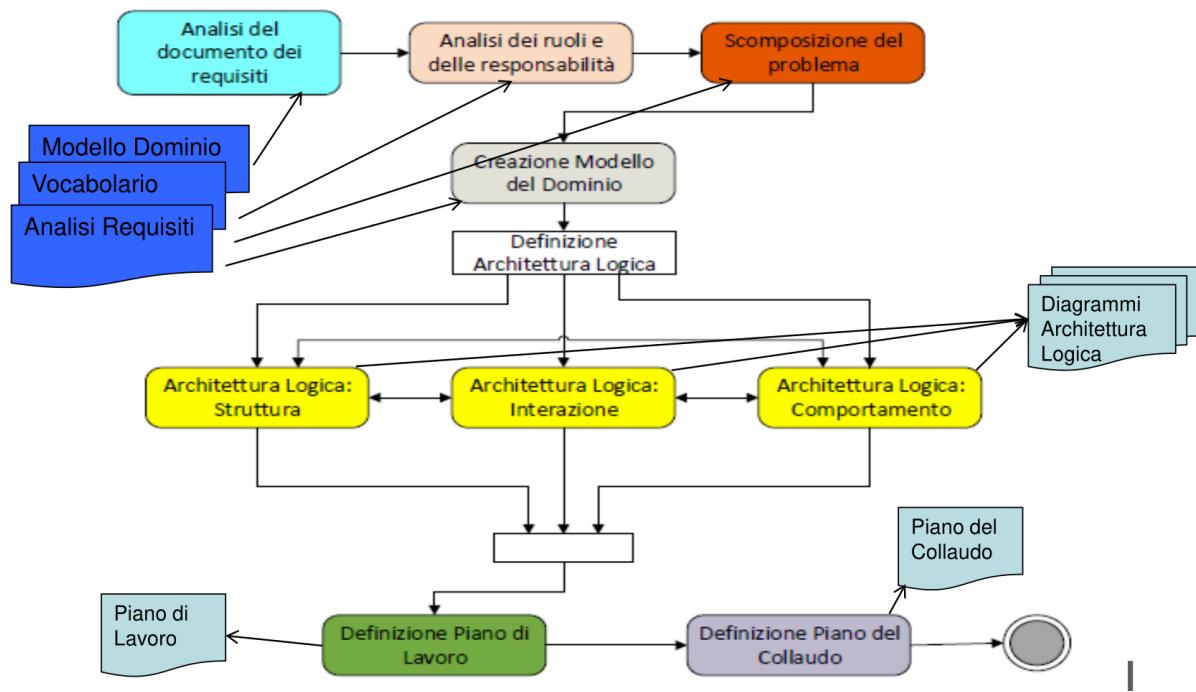
8. Definizione del Piano di Lavoro

Obiettivo: assegnare le responsabilità a ciascun membro del team di progetto, stabilire i vincoli temporali, impostare eventuali milestone, ...

9. Definizione del Piano del Collaudo

Obiettivo: definire i risultati attesi da ogni entità (classe, sottosistema) che

compare nell'Architettura Logica; scrivere le classi di test per verificare tali risultati



2.2.2.1 Architettura Logica

- L'**Architettura Logica** è un insieme di modelli UML costruiti per definire una struttura concettuale del problema robusta e modificabile
- Attraverso l'Architettura Logica l'analista descrive
 - **la struttura** (insieme delle parti)
 - **il comportamento atteso** (da tutto e da ciascuna parte)
 - **le interazioni**

così come possono essere dedotte dai requisiti, dalle caratteristiche del problema e del dominio applicativo **senza alcun riferimento alle tecnologie realizzative**

2.2.2.2 Piano di Lavoro

- Il **Piano di Lavoro** esprime l'articolazione delle attività in termini di risorse
 - umane
 - temporali
 - di elaborazione
 - ...

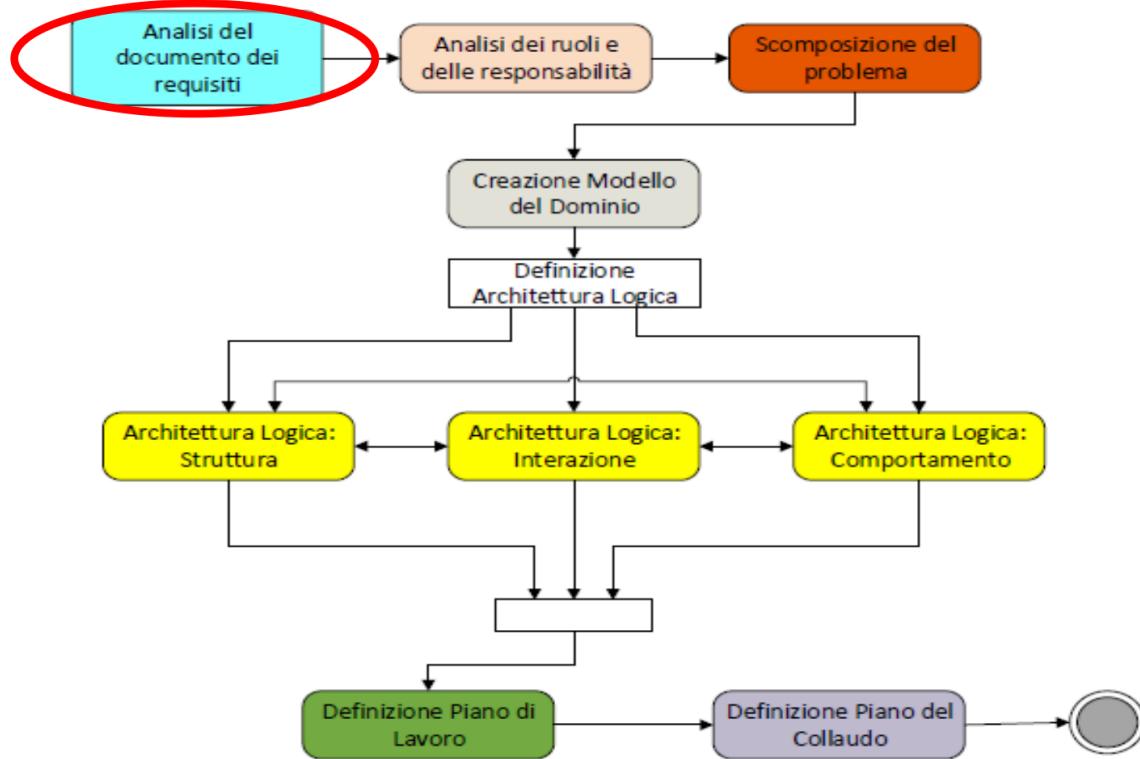
necessarie allo sviluppo del prodotto in base ai risultati dell'Analisi del Problema

2.2.2.3 Piano del Collaudo

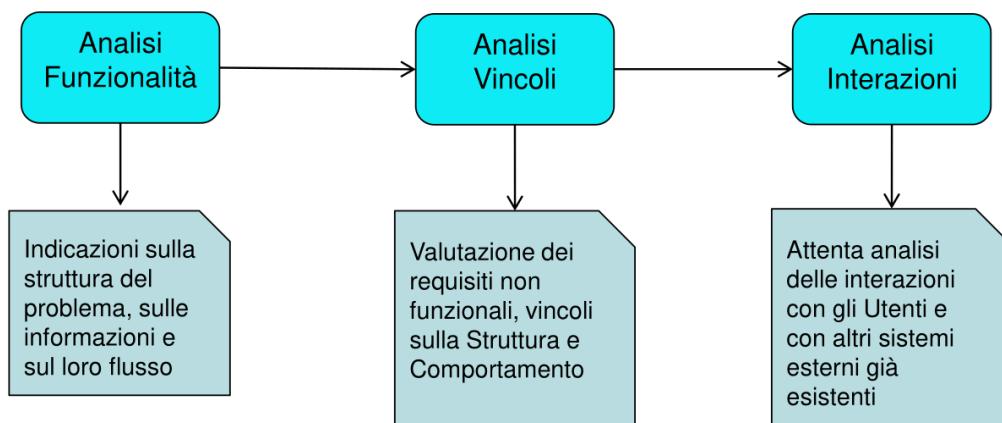
- Il **Piano del Collaudo** definisce l'insieme dei risultati attesi da ogni entità definita nell'Architettura Logica in relazione a specifiche sollecitazioni stimolo-risposta prevista
- Un modo per impostare il Piano del Collaudo è quello di fornire i test delle classi definite nell'Architettura Logica, così da pianificare già in questa fase i test di integrazione dei vari sottosistemi
- Ciò agevola il lavoro della successiva fase di progetto, riducendo il rischio di brutte sorprese in fase di integrazione delle sotto-parti
- Strumenti tipici per scrivere i test-case:
 - JUnit nel mondo Java

- ▶ NUnit(<http://nunit.org/>) nel mondo C#

2.2.3 Analisi Documento dei Requisiti



- Il Documento dei Requisiti evidenzia
 - le **funzionalità** e i **servizi** che dovranno essere sviluppati (requisiti funzionali)
 - i **vincoli** che dobbiamo tenere in considerazione (requisiti non funzionali)
 - il “**mondo esterno**” con cui si dovrà interagire (attori e sistemi esterni)
 - i “**rischi**” legati a possibili attacchi alla protezione, integrità e privacy dei dati (requisiti di sicurezza)
- Partendo da tale documento occorre applicare un’attenta analisi delle singole funzionalità, vincoli, interazioni e rischi



2.2.3.1 Analisi delle Funzionalità

- Per ogni funzionalità espressa nei Casi d’Uso vanno analizzati:
 - **il tipo, ovvero l’obiettivo della funzionalità** : memorizzazione dei dati, interazione con l’esterno, gestione/manipolazione dei dati
 - etichettiamo la funzionalità in modo da rendere evidente durante la creazione dell’Architettura Logica dove “posizionarla”
 - se la funzionalità è molto complessa potrebbe appartenere a più tipi differenti:marcare la funzionalità come “complessa”

- ▶ **le informazioni coinvolte**: analisi sistematica di tutte le informazioni sulle quali deve “operare” la funzionalità
 - In particolare individuare il tipo di dato (composto/singolo) e il grado di riservatezza richiesto
 - ▶ **il “flusso delle informazioni”**: quali sono le informazioni che
 - si ricevono in input: permette di dare indicazioni sulla validazione dell’input e se sono presenti vincoli sui valori ammessi
 - vanno prodotte in output: permette di valutare dall’esterno se la funzionalità si comporta come ci si aspetta
-

- Si possono predisporre diverse tabelle di analisi

Tabella Funzionalità

Funzionalità	Tipo	Grado Complessità	Requisiti Collegati
nome funzionalità come compare nei casi d’uso	tipo della funzionalità	indicare se complessa o semplice	Requisiti a cui è collegata la funzionalità

Tabella Informazioni / Flusso

Informazione	Tipo	Livello di riservatezza/ privacy	Input/Output	Vincoli
nome (o id)	composto/ semplice	grado di riservatezza richiesto	specificare se input o output	eventuali vincoli sui valori attesi

Se il grado di complessità è alto (Funzionalità complessa), allora bisognerà scomporla in sotto-funzionalità.

- La Tabella delle Funzionalità è una sola tabella per tutte le funzionalità
- La Tabella delle Informazioni/Flusso è una tabella per ogni funzionalità; è importante specificare ogni funzionalità, anche se prima si sono tralasciate funzionalità semplici

2.2.3.2 Esempio

- Tabella Funzionalità per il Villaggio Turistico

Funzionalità	Tipo	Grado Complessità	Requisiti Collegati
GestioneOspite	Memorizzazione dati, gestione dati	complessa	R1F, R2F, R3F, R4F, R5F, R7F, R8F, R10F, R12F, R13F
Login	Interazione esterno, gestione dati	semplice	R14F, R15F
NuovoMovimento	Memorizzazione dati, gestione dati	semplice	R11F
ElencoVendite	Gestione dati	semplice	R6F
ScritturaLog	Memorizzazione dati	semplice	R16F
AnalisiLog	Gestione dati	semplice	R17F

- Tabella Informazioni/Flusso per NuovoMovimento

Informazione	Tipo	Livello protezione / privacy	Input/output	Vincoli
Data	semplice	Protezione media	Input	Non più di 10 caratteri
Ora	semplice	Protezione media	Input	Non più di 5 caratteri
Tipo di acquisto	semplice	Protezione alta	Input	Non più di 400 caratteri
Importo addebitato	semplice	Protezione alta	Input	Non più di 30 caratteri
Identificativo GuestCard	semplice	Protezione molto alta	Input	Non più di 20 caratteri
Identificativo Punto Vendita	semplice	Protezione molto alta	Input	Non più di 20 caratteri
Identificativo Catena Punti di Vendita	semplice	Protezione molto alta	Input	Non più di 20 caratteri

2.2.3.3 Analisi dei Vincoli

- Per ogni requisito non funzionale vanno analizzati:
 - **il tipo, ovvero a quale categoria appartiene:** vincoli sulle performance, sui tempi di risposta, sulla scalabilità, sull'usabilità, sulla protezione dei dati, etc...
 - etichettiamo il requisito in modo da rendere evidente la categoria
 - alcuni requisiti potrebbero influenzare diversi aspetti del sistema, nel caso indicarli tutti, insieme anche al tipo di impatto (es: portano ad un peggioramento)
 - **le funzionalità coinvolte:** indicare le funzionalità che vengono influenzate dal requisito
- Si può predisporre una tabella

Tabella Vincoli

Requisito	Categorie	Impatto	Funzionalità
Requisito non funzionale	tipo	indicare il tipo di impatto	funzionalità coinvolte



Una sola tabella per tutti i vincoli

2.2.3.4 Esempio

- Tabella Vincoli per il Villaggio Turistico

Requisito	Categorie	Impatto	Funzionalità
Velocità ricerca dati (R1NF)	Tempo di risposta	Cercare di migliorare	GestioneOspite, NuovoMovimento, ElencoVendite, Login
Velocità memorizzazione dati (R3NF)	Tempo di risposta	Cercare di migliorare	GestioneOspite, NuovoMovimento, ElencoVendite
Facile navigabilità delle schermate (R2NF)	Usabilità	Cercare di migliorare	GestioneOspite, NuovoMovimento, ElencoVendite, Login
Protezione dei dati (R3NF, R4NF, R6NF, R7NF, R8NF, R12NF)	Sicurezza	Peggiorano tempo di risposta, migliorano la privacy dei dati	GestioneOspite, NuovoMovimento, ElencoVendite, Login
Controllo Accessi (R5NF, R8NF, R9NF, R11NF)	Sicurezza	Peggiorano tempo di risposta e usabilità, migliorano la privacy dei dati	GestioneOspite, NuovoMovimento, ElencoVendite, Login

2.2.3.5 Analisi delle Interazioni

- Vanno distinte le interazioni con gli **umani** da quelle con **sistemi esterni**
- Nel caso delle interazioni con gli umani
 - analizzare le eventuali interfacce identificate con il cliente nella fase di Analisi dei Requisiti, oppure delineare le possibili interfacce
 - individuare le maschere di inserimento/output dati
 - individuare le sole informazioni necessarie da mostrare in ogni maschera
 - creare un legame tra maschere - informazioni - funzionalità
- Si può predisporre una tabella

Tabella Maschere

Maschera	Informazioni	Funzionalità
Nome della maschera	indicare le informazioni gestite nella maschera	funzionalità coinvolte

Una sola tabella per tutte le maschere

2.2.3.6 Esempio

- Tabella Maschere per il Villaggio Turistico

Maschera	Informazioni	Funzionalità
Home Gestione	Messaggio benvenuto e scelta funzionalità	GestioneOspite
View Registrazione	Nome Ospite, Cognome Ospite, DataNascita, Indirizzo di residenza, Numero di telefono, Estremi del documento di identificazione, numero della stanza	GestioneOspite
View Apertura Credito	GuestCard, Identificativo OspitePagante	GestioneOspite
View Chiusura Credito	GuestCard, Identificativo OspitePagante, saldo	GestioneOspite
View Elenco Acquisti	Elenco dei Movimenti	GestioneOspite
Home PuntoVendita	Messaggio benvenuto e scelta funzionalità	NuovoMovimento
View Inserimento	Data, Ora, Tipo di acquisto, importo addebitato, GuestCard	NuovoMovimento
Home Catena PuntiVendita	Elenco dei report di vendita per ogni PuntoDiVendita	ElencoVendite
Home Log	Scelta del tipo di analisi o di visione di tutto il log di una parte del sistema	AnalisiLog
View Log	Data, Ora, Operazione eseguita, Messaggio	AnalisiLog
View Anomalie	Elenco delle anomalie	AnalisiLog
View Login	Username, password	Login

- Nel caso delle interazioni con sistemi esterni
 - analizzare ai morsetti i sistemi esterni con cui si dovrà interagire
 - individuare i protocolli di interazione con tali sistemi

Tabella Sistemi Esterni

Sistema	Descrizione	Protocollo di Interazione	Livello di Protezione
Nome del sistema	Descrizione del sistema e delle sue principali funzionalità	Specificare il protocollo di interazione richiesto	Specificare il livello di protezione garantito dal sistema

Una sola tabella per tutti i sistemi esterni

2.2.3.7 Esempio

- Tabella Sistemi Esterni per il Villaggio Turistico

Sistema	Descrizione	Protocollo di Interazione	Livello di Sicurezza
Gestione Personale (R16F/R9NF)	Sistema che si occupa della gestione del personale che lavora presso il Villaggio Turistico.	GestionePersonale mette a disposizione una funzionalità di controllo delle credenziali. Le credenziali devono essere inviate in modo sicuro e come risultato si ha una stringa che rappresenta il nome del ruolo assegnato al dipendente	Alto livello di sicurezza perché protegge i dati personali e sensibili dei dipendenti

2.2.4 Analisi Ruoli e Responsabilità

- Per ogni Attore individuato nei Casi d’Uso specificare
 - le responsabilità - cosa deve fare in ogni funzionalità
 - specificare le **informazioni a cui può accedere** relativamente a ciascuna funzionalità in cui è coinvolto con relativa indicazione del **tipo di accesso**
 - le maschere che può visualizzare
 - il suo livello di riservatezza - quale livello di riservatezza è necessario avere per poter ricoprire quel ruolo
 - la numerosità attesa - il numero di persone che possono giocare quel ruolo

2.2.4.1 Analisi dei Ruoli e Responsabilità

- Si possono predisporre alcune tabelle

Tabella Ruoli

Ruolo	Responsabilità	Maschere	Riservatezza	Numerosità
Nome del ruolo	Indicare le responsabilità assegnate	Indicare le maschere che devono essere visualizzate	Livello di riservatezza necessaria	Indicare la numerosità massima

Tabella Ruolo-Informazioni

Informazione	Tipo Accesso
Informazione	Specificare il tipo di accesso

Come prima, una sola tabella per tutti i ruoli e una tabella per ogni ruolo.

Non ha senso specificare una numerosità potenzialmente infinita, perché il numero delle persone che interagiranno con il sistema è una informazione che influenza come il sistema verrà progettato. Se la numerosità cambia nel tempo, probabilmente servirà riprogettare il sistema. È possibile inserire come specifica di numerosità “si progetti il sistema in modo che possa supportare il maggior numero di utenti”.

2.2.4.2 Esempio

- Tabella Ruoli per il Villaggio Turistico

Ruolo	Responsabilità	Maschere	Riservatezza	Numerosità
Operatore	Gestione di tutte le informazioni relative agli Ospiti del villaggio turistico	Home Gestione, View Registrazione, View Apertura Credito, View Chiusura Credito, View Elenco Acquisti, View Login	È richiesto un alto grado di riservatezza	Massimo 10 Operatori, considerando l'alternanza dei turni di lavoro e dei giorni di risposto
..

2.2.4.3 Esempio

- Tabella Operatore-Informazioni per il Villaggio Turistico

Informazione	Tipo di Accesso
Nome Ospite	Lettura/scrittura
Cognome Ospite	Lettura/scrittura
DataNascita	Lettura/scrittura
Indirizzo di residenza	Lettura/scrittura
Numero di telefono	Lettura/scrittura
Estremi del documento di identificazione utilizzato	Lettura/scrittura
Numero della stanza	Lettura/scrittura
CartaCredito	Scrittura
Identificativo GuestCard	Lettura/scrittura
DataInizioSoggiorno	Lettura/scrittura
DataFineSoggiorno	Lettura/Scrittura
Valuta	Lettura/scrittura
Saldo	Lettura
Username	Scrittura
Password	Scrittura

2.2.5 Scomposizione del Problema

- Il punto di partenza è l'Analisi delle Funzionalità
- Per ogni funzionalità marcata come “complessa” nella Tabella delle Funzionalità occorre valutare se sia possibile operare una scomposizione
 - ▶ la funzionalità può essere suddivisa in sotto-funzionalità più semplici?
 - ▶ quale “legame” sussiste tra le sotto-funzionalità?
 - ▶ quali informazioni devono “fluire” tra le sotto-funzionalità
- Si possono predisporre alcune tabelle

Tabella Scomposizione Funzionalità

Funzionalità	Scomposizione
nome funzionalità	elenco delle sotto-funzionalità

Tabella Sotto-Funzionalità

Sotto-Funzionalità	Sotto-Funzionalità	Legame	Informazioni
nome sotto-funzionalità	nome sotto-funzionalità	specificare il tipo di dipendenza/legame logico	specificare le informazioni scambiate

La tabella Sotto-Funzionalità elenca le dipendenze tra funzionalità.

- Tabella Scomposizione Funzionalità per il Villaggio Turistico

Funzionalità	Scomposizione
GestioneOspite	Registrazione, AperturaCredito, ChiusuraCredito, ElencoAcquisti

- Tabella Sotto-Funzionalità per il Villaggio Turistico

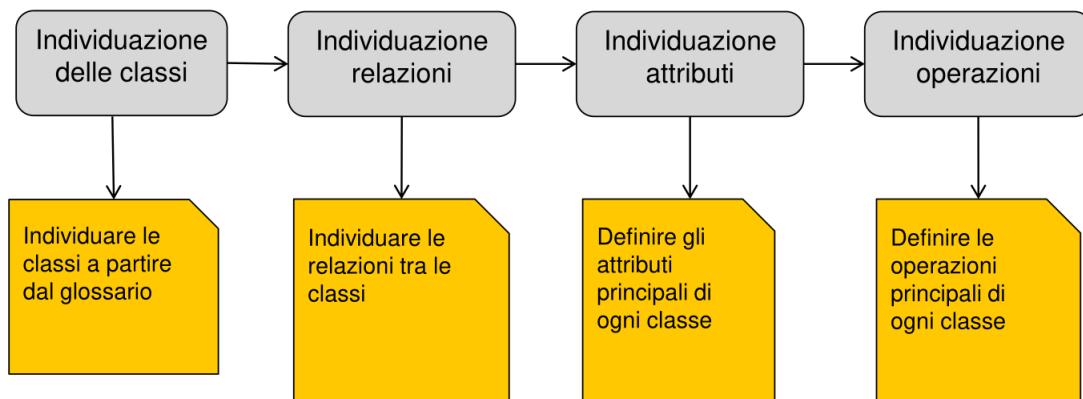
Sotto-funzionalità	Sotto-funzionalità	Legame	Informazioni
AperturaCredito	Registrazione	AperturaCredito dipende da Registrazione	Identificativo OspitePagante
ChiusuraCredito	AperturaCredito	Non si può chiudere un credito se non è mai stato aperto	Identificativo OspitePagante
ElencoAcquisti	AperturaCredito	Non si può mostrare l'elenco acquisti se non è stato aperto prima il credito	Identificativo OspitePagante

2.2.6 Creazione Modello del Dominio

- Il punto di partenza è costituito dall'insieme di:
 - [glossario](#) definito nell'Analisi dei Requisiti
 - tabelle informazioni/flusso
- Sulla base di questi si costruisce il diagramma delle classi che rappresenta il Modello del Dominio
- Tale modello poi sarà riusato nell'Architettura Logica come modello dei dati

- Occorre tenere presente che non tutti i vocaboli elencati nel glossario diventeranno classi, si devono infatti evitare:
 - ridondanze: classi uguali ma con diverso nome
 - costruzione di classi a partire da termini ambigui nel glossario
 - nomi che si riferiscono a eventi o operazioni
 - nomi appartenenti al meta-linguaggio del processo come, per esempio, requisiti e sistema
 - nomi al di fuori dell'ambito del sistema
 - nomi che possono essere attributi
- Individuare
 - **Oggetti e classi** rilevanti per il problema che si sta analizzando
 - Limitarsi esclusivamente a quelle classi che fanno parte del vocabolario del dominio del problema, non dell'applicazione in se, quindi evitare, ad esempio, di pensare a “che tipo di classe Java devo sviluppare”
 - **Relazioni tra le classi**
 - Per ogni classe
 - **Attributi**
 - **Operazioni fondamentali** cioè servizi forniti all'esterno

Pensare sempre al “*cosa*” non al “*come*”; il “*come*” non è l’oggetto di questa fase.



Questo schema non è per forza in successione, nel senso che se, mentre stiamo definendo le operazioni di una classe, mi accorgo che c’è bisogno di un attributo, è possibile aggiungerlo.

2.2.6.1 Individuazione delle Classi

- Dal glossario eliminare i nomi che sicuramente
 - non si riferiscono a classi
 - indicano attributi (dati di tipo primitivo)
 - indicano operazioni
- Scegliere un solo termine significativo se più parole indicano lo stesso concetto (sinonimi)
- Il nome della classe deve essere un nome familiare
 - all’utente o all’esperto del dominio del problema
 - non allo sviluppatore!
- **Attenzione agli aggettivi e agli attributi**, possono

- ▶ indicare oggetti diversi
- ▶ indicare usi diversi dello stesso oggetto
- ▶ essere irrilevanti
- Ad esempio:
 - ▶ “Studente bravo” potrebbe essere irrilevante
 - ▶ “Studente fuori corso” potrebbe essere una nuova classe
- **Attenzione alle frasi passive, impersonali o con soggetti fuori dal sistema:**
 - ▶ devono essere rese attive ed esplicite, perché potrebbero mascherare entità rilevanti per il sistema in esame
- Individuare **Attori** con cui il sistema in esame deve interagire
 - ▶ **Persone**: Docente, Studente, Esaminatore, Esaminando, ...
 - ▶ **Sistemi esterni**: ReteLocale, Internet, DBMS, ...
 - ▶ **Dispositivi**: attuatori, sensori, ...
- Individuare **modelli e loro elementi specifici**:
 - ▶ Insegnamento - “Ingegneria del Software T”
 - ▶ CorsoDiStudio - “Ingegneria Informatica”
 - ▶ Facoltà - “Ingegneria”
- Individuare **cose tangibili**, cioè oggetti reali appartenenti al dominio del problema
 - ▶ Banco, LavagnaLuminosa, Schermo, Computer, ...
- Individuare **contenitori** (fisici o logici) di altri oggetti
 - ▶ Facoltà, Dipartimento, Aula, SalaTerminali, ...
 - ▶ ListaEsame, CommissioneDiLaurea, OrdineDegliStudi, ...
- Individuare **eventi o transazioni** che il sistema deve gestire e memorizzare
 - ▶ possono avvenire in un certo istante (es., una vendita) o
 - ▶ possono durare un intervallo di tempo (es., un affitto)
 - ▶ Appello, EsameScritto, Registrazione, AppelloDiLaurea, ...
- Per determinare se includere una classe nel modello, porsi le seguenti domande:
 - ▶ il sistema deve interagire in qualche modo con gli oggetti della classe?
 - **utilizzare informazioni** (attributi) contenute negli oggetti della classe
 - **utilizzare servizi** (operazioni) offerti dagli oggetti della classe
 - ▶ quali sono le responsabilità della classe nel contesto del sistema?
- Attributi e operazioni devono essere applicabili a tutti gli oggetti della classe
- Se esistono
 - ▶ attributi con un valore ben definito solo per alcuni oggetti della classe
 - ▶ operazioni applicabili solo ad alcuni oggetti della classe

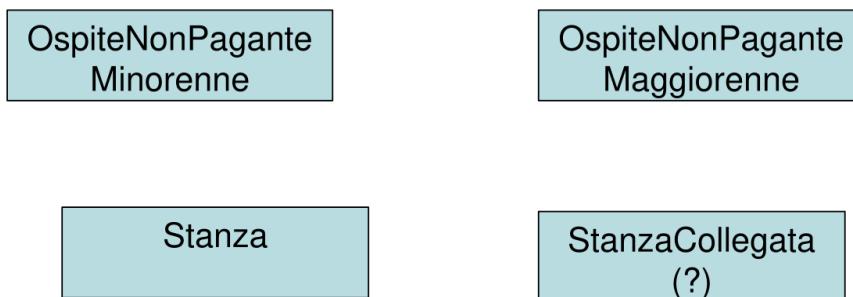
siamo in presenza di **ereditarietà**

- Esempio: dopo una prima analisi, la classe Studente potrebbe contenere un attributo booleano inCorso, ma un’analisi più attenta potrebbe portare alla luce la gerarchia:
 - ▶ Studente: StudenteInCorso, StudenteFuoriCorso

2.2.6.2 Individuazione delle Classi: Villaggio Turistico

Voce	Definizione	Sinonimi
Villaggio Turistico	Luogo dove si effettua una vacanza	
Ospite	Persona che è in vacanza nel Villaggio Turistico, senza fare distinzione se Pagante o non Pagante	Cliente
OspitePagante	Persona in vacanza nel villaggio turistico a cui sono associate una o più stanze. Persona che effettua il saldo del conto	Cliente
OspiteNonPagante	Persona in vacanza nel villaggio turistico a cui è associata una stanza. Non effettua pagamenti ed è sempre associato a un OspitePagante	
OspiteNonPagante Maggiorenne	Persona in vacanza nel villaggio turistico a cui è associata una stanza. Ha raggiunto la maggiore età. Gli viene consegnata una GuestCard, ma questa viene associata all'OspitePagante.	
OspiteNonPagante Minorenne	Persona in vacanza nel villaggio turistico a cui è associata una stanza. Non ha raggiunto la maggiore età.	
Stanza	Ambiente fisico in cui gli Ospiti dormono e tengono i loro beni. Ha un certo numero di posti disponibili	camera
StanzaCollegata	Ambiente fisico in cui gli Ospiti dormono e tengono i loro beni. È logicamente collegata ad un'altra stanza	camera

- Villaggio Turistico è in verde perché è una classe che non viene creata, dato che non serve; se, invece, avessimo avuto un progetto che prevedeva molteplici villaggi turistici, allora avrebbe avuto senso definire la classe VillaggioTuristico
- I nomi in rosso sono delle classi, tra l'altro con una probabile gerarchia (alcune)
- Forse Stanza e StanzaCollegata sono una gerarchia



2.2.6.3 Individuazione delle Relazioni

- La maggior parte delle classi (degli oggetti) **interagisce** con altre classi (altri oggetti) in vari modi
- In ogni tipo di relazione, esiste un cliente C che dipende da un fornitore di servizi F
 - C ha bisogno di F per lo svolgimento di alcune funzionalità che C non è in grado di effettuare autonomamente
 - Conseguenza per il corretto funzionamento di C è indispensabile il corretto funzionamento di F

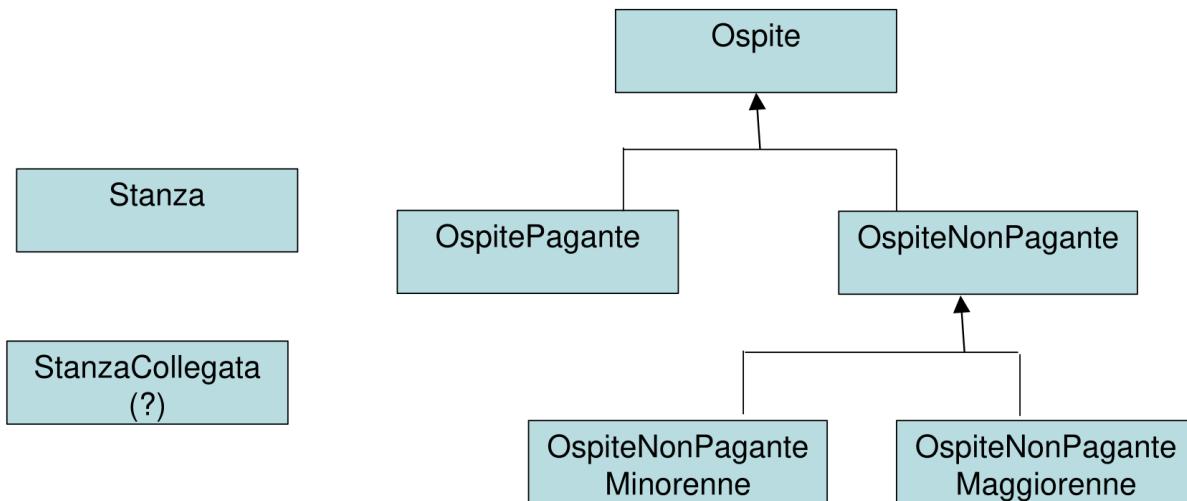
Tipo di relazione	Cliente	Fornitore
Ereditarietà	sottoclasse	superclasse
Associazione (composizione / aggregazione)	contenitore	contenuto
Dipendenza	classe dipendente (source)	classe da cui si dipende (target)

- Nell'associazione il fornitore è il contenuto e il cliente è il contenitore, perché il contenitore contenendo il contenuto può usare i servizi del contenuto

2.2.6.4 Individuazione dell'Ereditarietà

- L'ereditarietà deve rispecchiare una tassonomia effettivamente presente nel dominio del problema
 - Non usare l'ereditarietà dell'implementazione (siamo ancora in fase di analisi!)
 - Non usare l'ereditarietà solo per riunire caratteristiche comuni
 - ad es., Studente e Dipartimento hanno entrambi un indirizzo, ma non per questo c'è ereditarietà!

2.2.6.5 Ereditarietà: Villaggio Turistico



- Tutte le ereditarietà di questo caso sono complete e disgiunte; cioè non esistono ospiti che non siano paganti e non paganti e un ospite è o pagante o non pagante, non esistono ospiti non paganti che non siano minorenni e maggiorenni e un ospite non pagante è o minorenne o maggiorenne

2.2.6.6 Individuazione delle Associazioni

- Un'associazione rappresenta una relazione strutturale tra due istanze di classi diverse o della stessa classe
- Un'associazione può
 - Rappresentare un contenimento logico (**aggregazione**)
 - Una lista d'esame contiene degli studenti
 - Rappresentare un contenimento fisico (**composizione**)
 - Un triangolo contiene tre vertici

- Non rappresentare un reale contenimento
 - Una fattura si riferisce a un cliente
 - Un evento è legato a un dispositivo

2.2.7 Individuazione delle Associazioni

- Aggregazione

Un oggetto x di classe X è associato a (contiene) un oggetto y di classe Y in modo non esclusivo x può condividere y con altri oggetti anche di tipo diverso (che a loro volta possono contenere y)

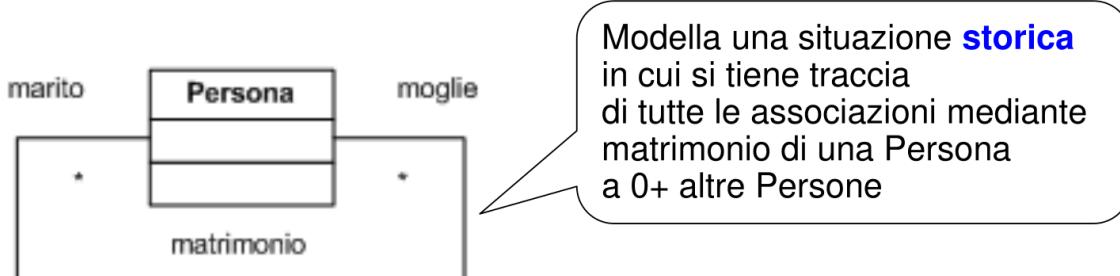
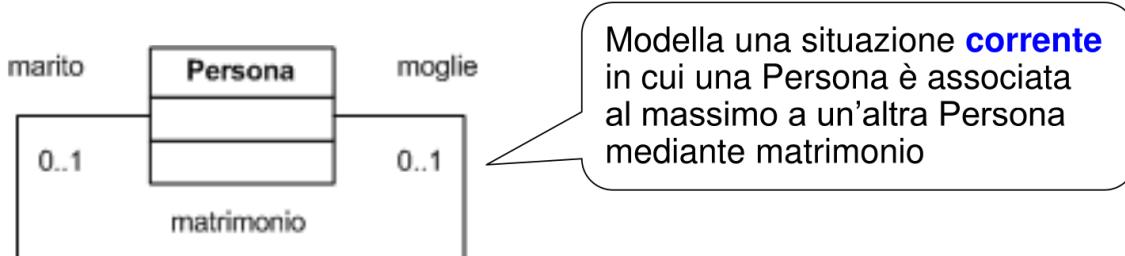
- Una lista d'esame contiene degli studenti
 - Uno studente può essere contemporaneamente in più liste d'esame
 - La cancellazione della lista d'esame non comporta l'eliminazione "fisica" degli studenti in lista

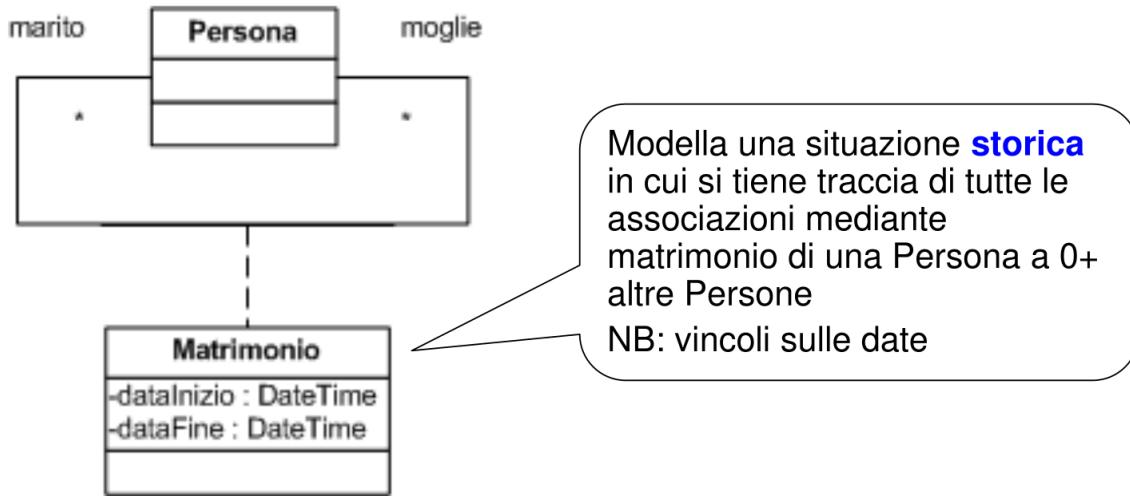
- Composizione

Un oggetto x di classe X è associato a (contiene) un oggetto y di classe Y **in modo esclusivo** y esiste solo in quanto contenuto in x

- Un triangolo contiene tre punti (i suoi vertici)
 - L'eliminazione del triangolo comporta l'eliminazione dei tre punti
 - Se la distruzione del contenitore comporta la distruzione degli oggetti contenuti, si tratta di composizione, altrimenti si tratta di aggregazione
 - Attenzione alle associazioni molti a molti possono nascondere una classe (classe di associazione) del tipo “evento da ricordare”
 - Ad esempio,
 - la connessione “matrimonio” tra Persona e Persona può nascondere una classe Matrimonio, che lega due Persone
 - la connessione “possiede” tra Proprietario e Veicolo può nascondere una classe CompraVendita, che lega un Proprietario a un Veicolo

2.2.7.1 1° Esempio di Associazione

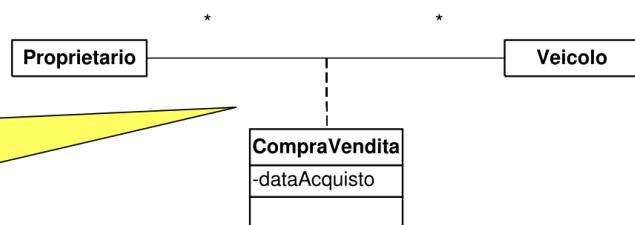
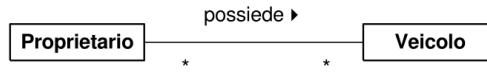




Quindi, quando l'associazione ha delle informazioni, o più raramente del comportamento, si introduce la classe d'associazione.

2.2.7.2 2 ° Esempio di Associazione

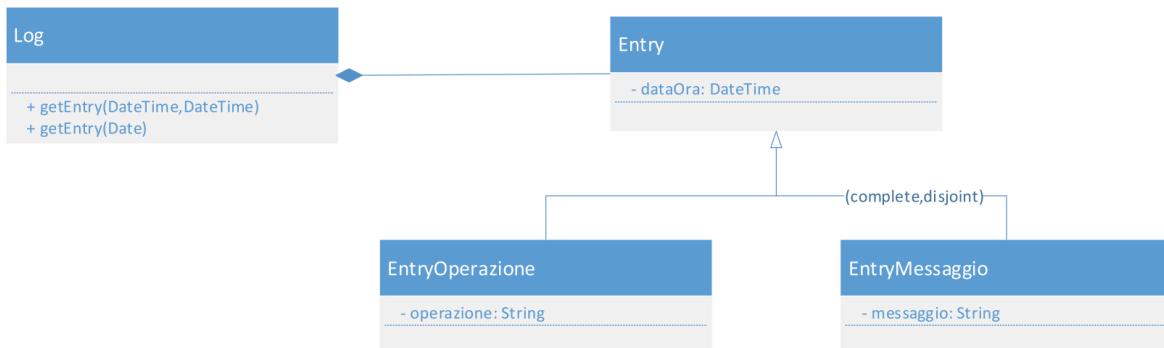
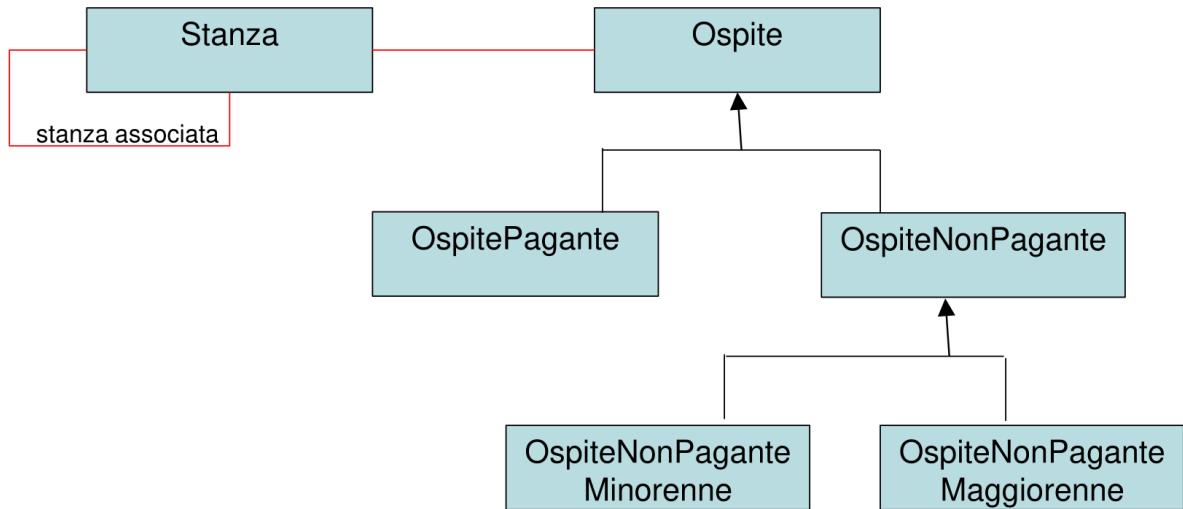
- Un proprietario può possedere molti veicoli
- Un veicolo può essere di molti proprietari
 - in tempi successivi
 - in comproprietà



Attenzione alle molteplicità: nel modello delle classi di analisi va bene, ma a livello di istanza si introduce il vincolo extra che ci può essere solo un'istanza della classe di associazione tra ogni coppia di oggetti associati

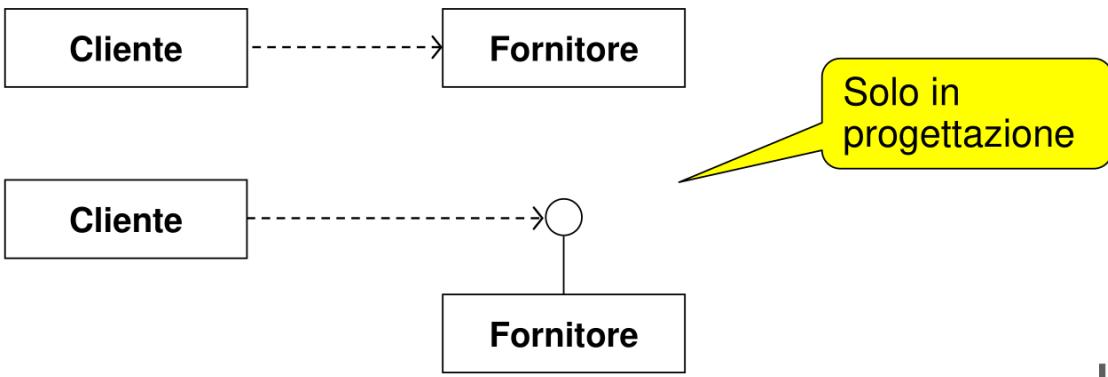
- Qui c'è un vincolo particolare, che è quello di tenere in conto del caso in cui una persona compra un veicolo da solo e, successivamente, vuole averlo in comproprietà
- Da notare che ci può essere solo un'istanza della classe di associazione per ogni coppia di oggetti associati

2.2.7.3 Associazioni: Villaggio Turistico



2.2.7.4 Individuazione Collaborazioni

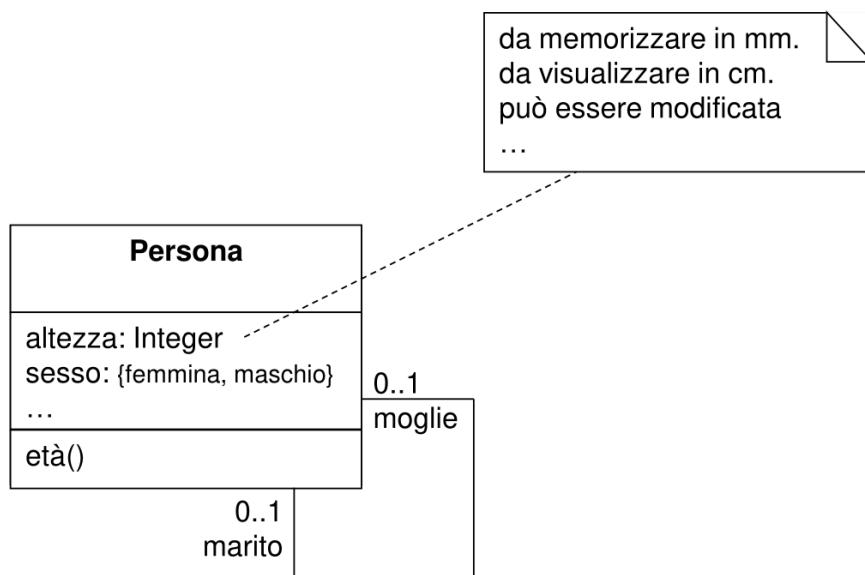
- Una classe A è in relazione USA con una classe B (A USA B) quando A ha bisogno della collaborazione di B per lo svolgimento di alcune funzionalità che A non è in grado di effettuare autonomamente
 - Un'operazione della classe A ha bisogno come argomento di un'istanza della classe B
 - `void fun1(B b) { ... usa b ... }`
 - Un'operazione della classe A restituisce un valore di tipo B
 - `B fun2(...) { B b; ... return b; }`
 - Un'operazione della classe A utilizza un'istanza della classe B (ma non esiste un'associazione tra le due classi)
 - `void fun3(...) { B b = new B(...); ... usa b ... }`
- La relazione non è simmetrica: A dipende da B, ma B non dipende da A
- Evitare situazioni in cui una classe, tramite una catena di relazioni USA, alla fine dipende da se stessa



2.2.7.5 Individuazione degli Attributi

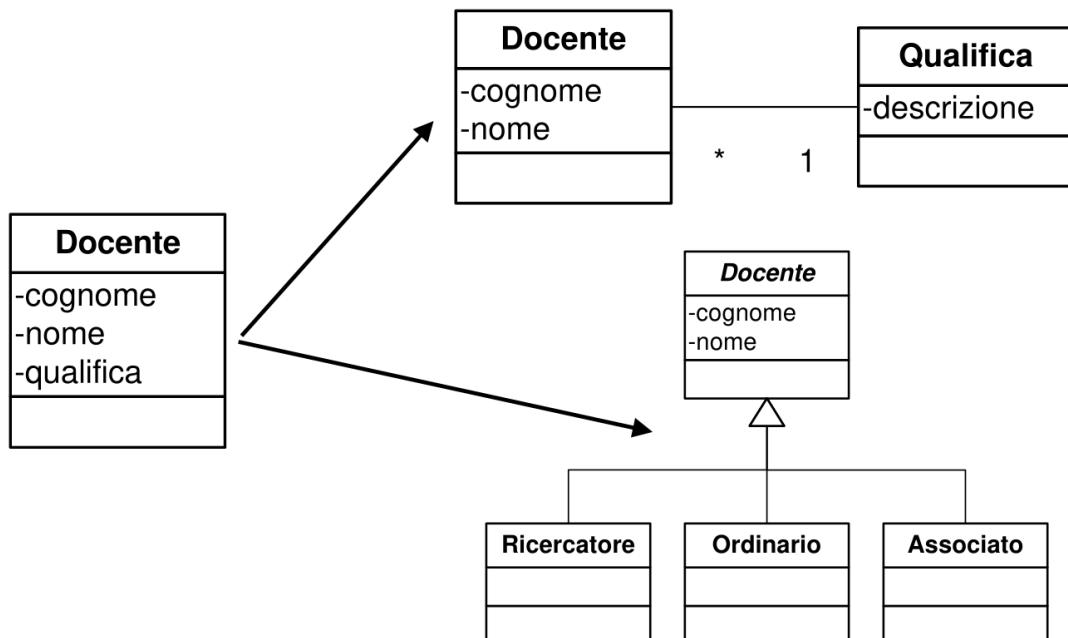
- Ogni attributo modella una proprietà atomica di una classe
 - Un valore singolo
 - Una descrizione, un importo, ..
 - Un gruppo di valori strettamente collegati tra loro
 - Un indirizzo, una data, ..
- Proprietà non atomiche di una classe devono essere modellate come associazioni
- A tempo di esecuzione, in un certo istante, ogni oggetto avrà un valore specifico per ogni attributo della classe di appartenenza: informazione di stato
- La base di partenza per la ricerca degli attributi sono le Tabelle di Informazione/ Flusso
- Il nome dell'attributo
 - deve essere un nome familiare
 - all'utente o all'esperto del dominio del problema
 - non allo sviluppatore!
 - non deve essere il nome di un valore ("qualifica" sì, "ricercatore" no)
 - deve iniziare con una lettera minuscola
- Esempi
 - cognome, dataDiNascita, annoDiImmatricolazione
- Esprimere tutti i vincoli applicabili all'attributo
 - Tipo (semplice o enumerativo)
 - Opzionalità
 - Valori ammessi
 - dominio, anti-dominio, univocità
 - Vincoli di creazione
 - valore iniziale di default, immodificabilità del valore (readonly), etc.
 - Vincoli dovuti ai valori di altri attributi
 - Unità di misura, precisione
- Esprimere tutti i vincoli applicabili all'attributo
 - **Visibilità** (opzionale in fase di analisi) Attenzione: gli attributi membro devono essere sempre privati!
 - Appartenenza alla classe (e non all'istanza) Attributi (e associazioni) possono essere di classe, cioè essere unici nella classe
 - numIstanze: int = 0

- I vincoli possono essere scritti
 - Utilizzando direttamente UML
 - Utilizzando Object Constraint Language (OCL)
 - Come testo in formato libero in un commento UML



- Attenzione: nel caso di attributi con valore booleano “vero o falso”, “sì o no”, il nome dell’attributo potrebbe essere uno dei valori di un’enumerazione
 - Ad esempio: tassabile (sì o no) potrebbe diventare tipoTassazione{ tassabile, esente, ridotto, ... }
- Attenzione: attributi
 - con valore “non applicabile” o
 - con valore opzionale o
 - a valori multipli (enumerazioni)

possono nascondere ereditarietà o una nuova classe



- Attenzione: nel caso di attributi calcolabili (ad esempio, età), specificare sempre l’operazione di calcolo mai l’attributo

- se memorizzare oppure no un attributo calcolabile è una decisione progettuale, un compromesso tra tempo di calcolo e spazio di memoria
- Applicare l'ereditarietà:
 - Posizionare attributi e associazioni più generali più in alto possibile nella gerarchia
 - Posizionare attributi e associazioni specializzati più in basso

2.2.7.6 Individuazione degli Attributi: Villaggio Turistico

Informazione	Voce	Definizione
Nome Ospite	Stanza	Ambiente fisico in cui gli Ospiti dormono e tengono i loro bene. Ha un certo numero di posti disponibili
Cognome Ospite	StanzaCollegata	Ambiente fisico in cui gli Ospiti dormono e tengono i loro bene. È logicamente collegata ad un'altra stanza

Ospite

```

- nome: String
- cognome: String
- indirizzo: String
- telefono
- dataNascita: Date
- documentoID: String
- inizioSoggiorno: DateTime
- fineSoggiorno: DateTime
  
```

Informazione	Voce	Definizione
Nome Ospite	Stanza	Ambiente fisico in cui gli Ospiti dormono e tengono i loro bene. Ha un certo numero di posti disponibili
Cognome Ospite	StanzaCollegata	Ambiente fisico in cui gli Ospiti dormono e tengono i loro bene. È logicamente collegata ad un'altra stanza

Ospite

```

- nome: String
- cognome: String
- indirizzo: String
- telefono
- dataNascita: Date
- documentoID: String
- inizioSoggiorno: DateTime
- fineSoggiorno: DateTime
  
```

Stanza

```

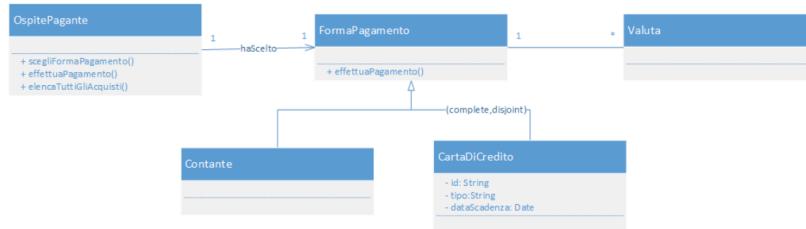
- numero: Integer
- tipo: String
- occupanti: Integer
  
```

- GuestCard una classe, l'identificativo sarà mappato con una associazione
- Saldo nasconde una nuova classe, associata alle funzionalità di Apertura e Chiusura Credito → L'ospite ha un conto aperto presso il Villaggio Turistico
- Movimento è relativo all'uso della GuestCard

Informazione
Nome Ospite
Cognome Ospite
DataNascita
Indirizzo di residenza
Numero di telefono
Estremi del documento di identificazione utilizzato
CartaCredito
Numeri della stanza
Data inizio soggiorno
Data fine soggiorno
Identificativo GuestCard
Movimento
Valuta
Saldo

Voce	Definizione
Stanza	Ambiente fisico in cui gli Ospiti dormono e tengono i loro bene. Ha un certo numero di posti disponibili
StanzaCollegata	Ambiente fisico in cui gli Ospiti dormono e tengono i loro bene. È logicamente collegata ad un'altra stanza

Carta di Credito e Valuta sono relative alle funzionalità di pagamento scelta e vanno modellate con delle classi



2.2.7.7 Individuazione delle Operazioni

- Il nome dell'operazione
 - deve appartenere al vocabolario standard del dominio del problema
 - potrebbe essere un verbo
 - all'imperativo (scrivi, esegui, ...) o
 - in terza persona (scrive, esegue, ...)
 - in modo consistente in tutto il sistema
- Operazioni standard
 - Operazioni che tutti gli oggetti hanno per il semplice fatto di esistere e di avere degli attributi e delle relazioni con altri oggetti (costruttore, accessori, ...)
 - Sono implicate e, di norma, non compaiono nel diagramma delle classi di analisi
- Altre operazioni
 - Devono essere determinate
 - servizi offerti agli altri oggetti
 - Compaiono nel diagramma delle classi di analisi
- Classi contenitori
 - Operazioni standard - aggiungi, rimuovi, conta, itera, ...
 - Altre operazioni - riguardano l'insieme degli oggetti, non il singolo oggetto
 - Calcoli da effettuare sugli oggetti contenuti Es: calcolaSulleParti(), totalizza()
 - Selezioni da fare sugli oggetti contenuti Es: trovaPartiSpecifiche()
 - Operazioni del tipo Es: eseguiUnAzioneSuTutteLeParti()
- Distribuire in modo bilanciato le operazioni nel sistema
- Mettere ogni operazione “vicino” ai dati a essa necessari
- Applicare l'ereditarietà
 - Posizionare le operazioni più generali più in alto possibile nella gerarchia
 - Posizionare le operazioni specializzate più in basso
- Descrivere tutti i vincoli applicabili all'operazione
 - Parametri formali, loro tipo e significato
 - Pre-condizioni, post-condizioni, invarianti di classe

- ▶ Eccezioni sollevate
- ▶ Eventi che attivano l'operazione
- ▶ Applicabilità dell'operazione
- ▶ ...
- **Pre-condizione**
Espressione logica riguardante le aspettative sullo stato del sistema prima che venga eseguita un'operazione
 - ▶ Esplicita in modo chiaro che è responsabilità della procedura chiamante controllare la correttezza degli argomenti passati
 - ▶ Ad esempio, per l'operazione **CalcolaRadiceQuadrata(valore)**, la pre-condizione potrebbe essere: “**valore ≥ 0**”

- **Post-condizione**
Espressione logica riguardante le aspettative sullo stato del sistema dopo l'esecuzione di un'operazione
 - ▶ Ad esempio, per l'operazione **CalcolaRadiceQuadrata(valore)**, la post-condizione potrebbe essere: **valore == risultato * risultato**

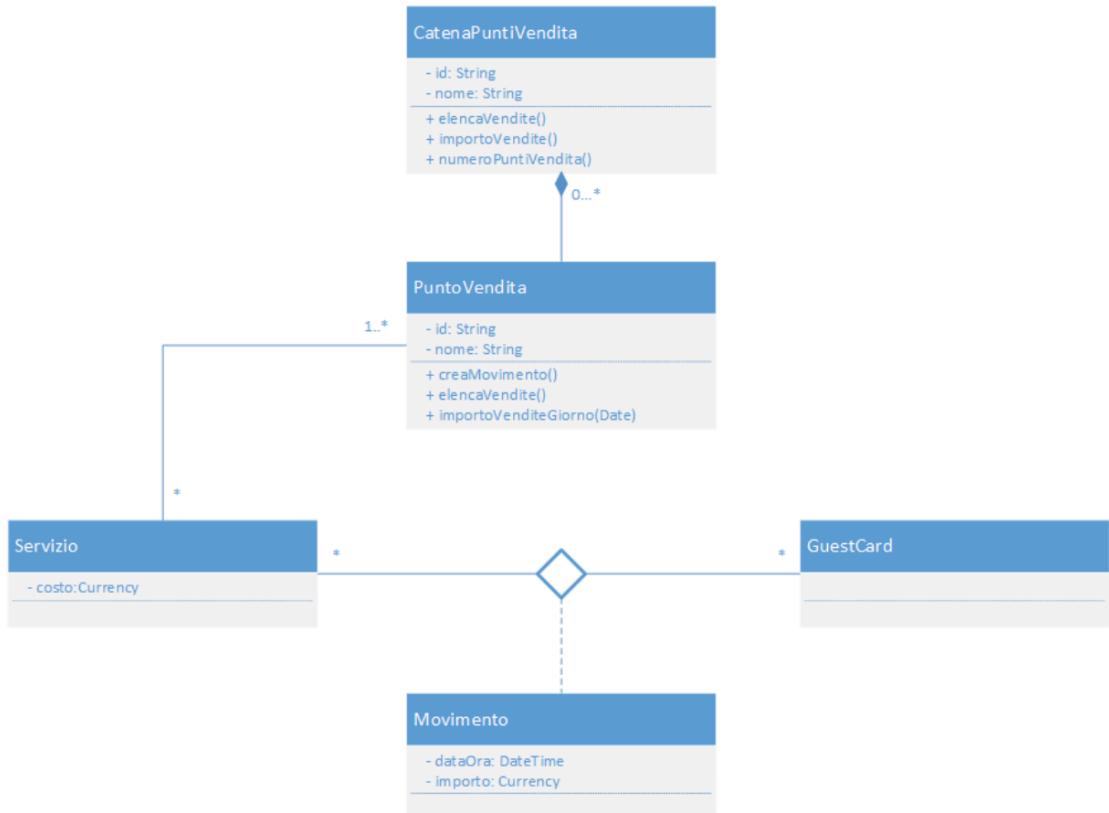
- **Invariante di classe**
Vincolo di classe (espressione logica) che deve essere sempre verificato
 - ▶ sia all'inizio
 - ▶ sia alla fine

di tutte le operazioni pubbliche della classe Può non essere verificato solo durante l'esecuzione dell'operazione

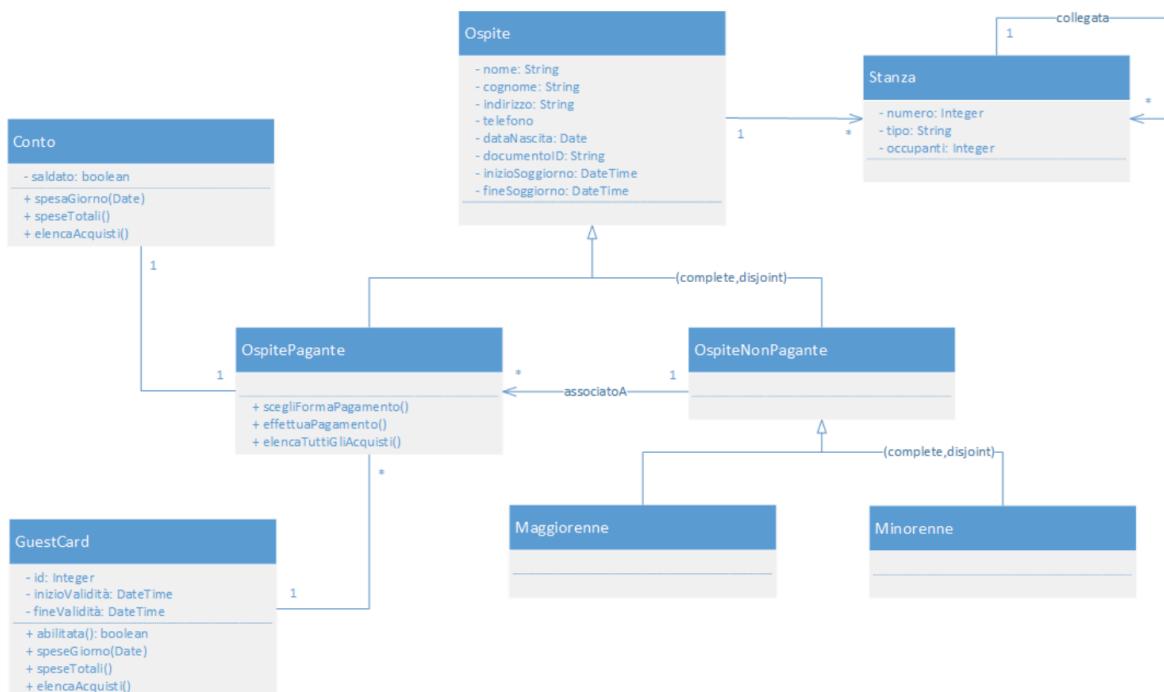
- **ECCEZIONE** Si verifica quando un'operazione
 - ▶ viene invocata nel rispetto delle sue pre-condizioni
 - ▶ ma non è in grado di terminare la propria esecuzione nel rispetto delle post-condizioni

2.2.7.8 Esempio

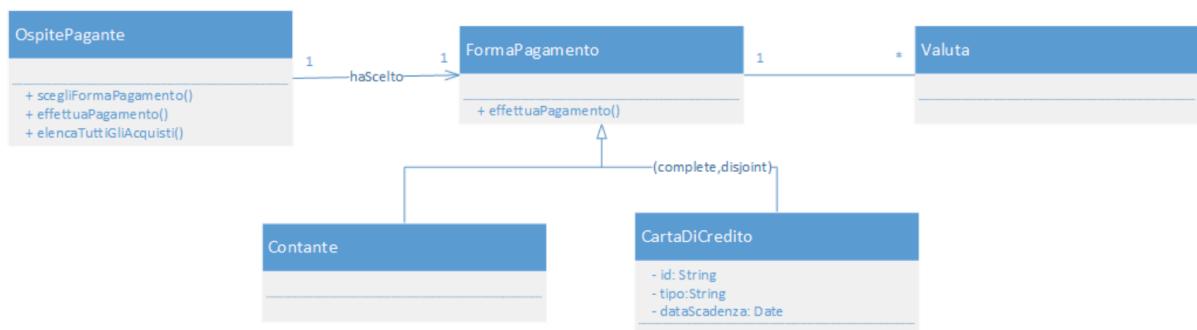
Modello del Dominio “Vendita Servizi” per il Villaggio Turistico



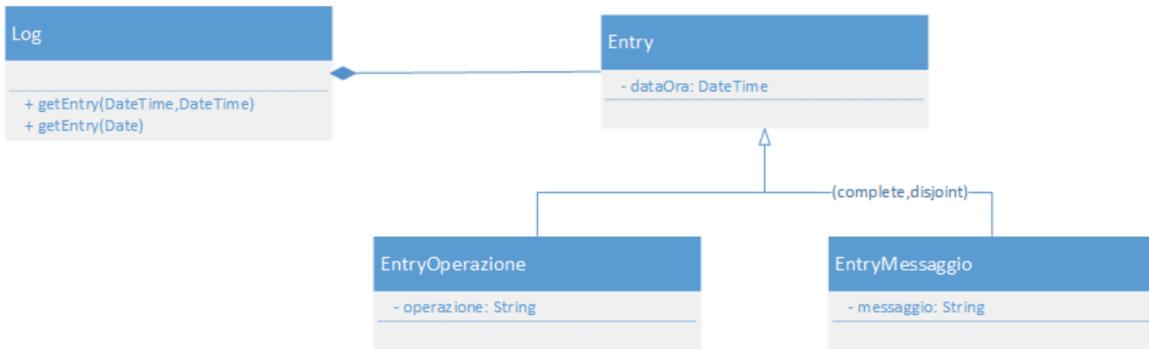
Modello del Dominio “Ospite” per il Villaggio Turistico



Modello del Dominio “Pagamenti” per il Villaggio Turistico



Modello del Dominio “Log” per il Villaggio Turistico



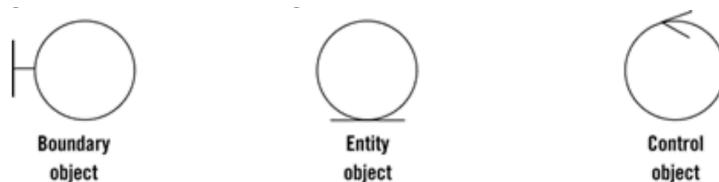
2.2.8 Architettura Logica: Struttura

N.B. Le parti “Struttura”, “Interazione” e “Comportamento” dell’architettura logica vanno sviluppate contestualmente (in contemporanea?).

- La parte strutturale dell’Architettura Logica dovrebbe essere composta di due tipi differenti di diagrammi UML
 - [Diagramma dei Package](#) che fornisce una visione di alto livello dell’architettura
 - [Diagramma delle classi](#) (uno o più diagrammi in base alla complessità) che fornisce una visione più dettagliata del contenuto dei singoli package
- Sarebbe opportuno organizzare sin da subito l’Architettura Logica usando un pattern architettonico chiamato **Boundary-Control-Entity** (BCE)

2.2.8.1 BCE

- BCE è un pattern architettonico che suggerisce di basare l’architettura di un sistema sulla partizione sistematica degli use case in oggetti di tre categorie:
 - **informazione**
 - **presentazione**
 - **controllo**



Noi useremo una convenzione più semplice → coloriamo package e classi in modo diverso

- A ciascuna di queste dimensioni corrisponde uno specifico insieme di classi
- Tale pattern è stato introdotto anche in RUP e sono state adottate icone ben particolari
- BCE è un pattern architettonico che suggerisce di basare l’architettura di un sistema sulla partizione sistematica degli use case in oggetti di tre categorie:
 - **informazione**
 - **presentazione**
 - **controllo**

- **Entity:** è la dimensione relativa alle entità cui corrisponde l'insieme delle classi che includono funzionalità relative alle informazioni che caratterizzano il problema; sostanzialmente ci dice quali sono le entità di interesse nel dominio del problema
 - costituiscono gran parte del modello del dominio
- **Boundary:** è la dimensione relativa alle funzionalità che dipendono dall'ambiente esterno cui corrisponde l'insieme delle classi che incapsulano l'interfaccia del sistema verso il mondo esterno
- **Control:** è la dimensione relativa agli enti che incapsulano il controllo
 - il loro compito è di fare da *collante* tra le interfacce e le entità
- Impostare l'architettura di un sistema software distinguendo tra *boundary*, *control* ed *entity* costituisce un solido punto di partenza per l'organizzazione dell'Architettura Logica di molte applicazioni
- L'analista tende ad affrontare la complessità dei problemi partizionando i problemi stessi in sotto-problemi
- È del tutto logico che un analista eviti di associare alle entità di un dominio
 - sia le funzionalità di una specifica applicazione
 - sia le funzionalità tipiche della interazione con l'utente
- La conseguenza è che l'architettura di un sistema software risulta quasi fisiologicamente articolata in una sequenza di livelli (*layer*) verticali che viene tipicamente mantenuta anche in fase di progetto e implementazione

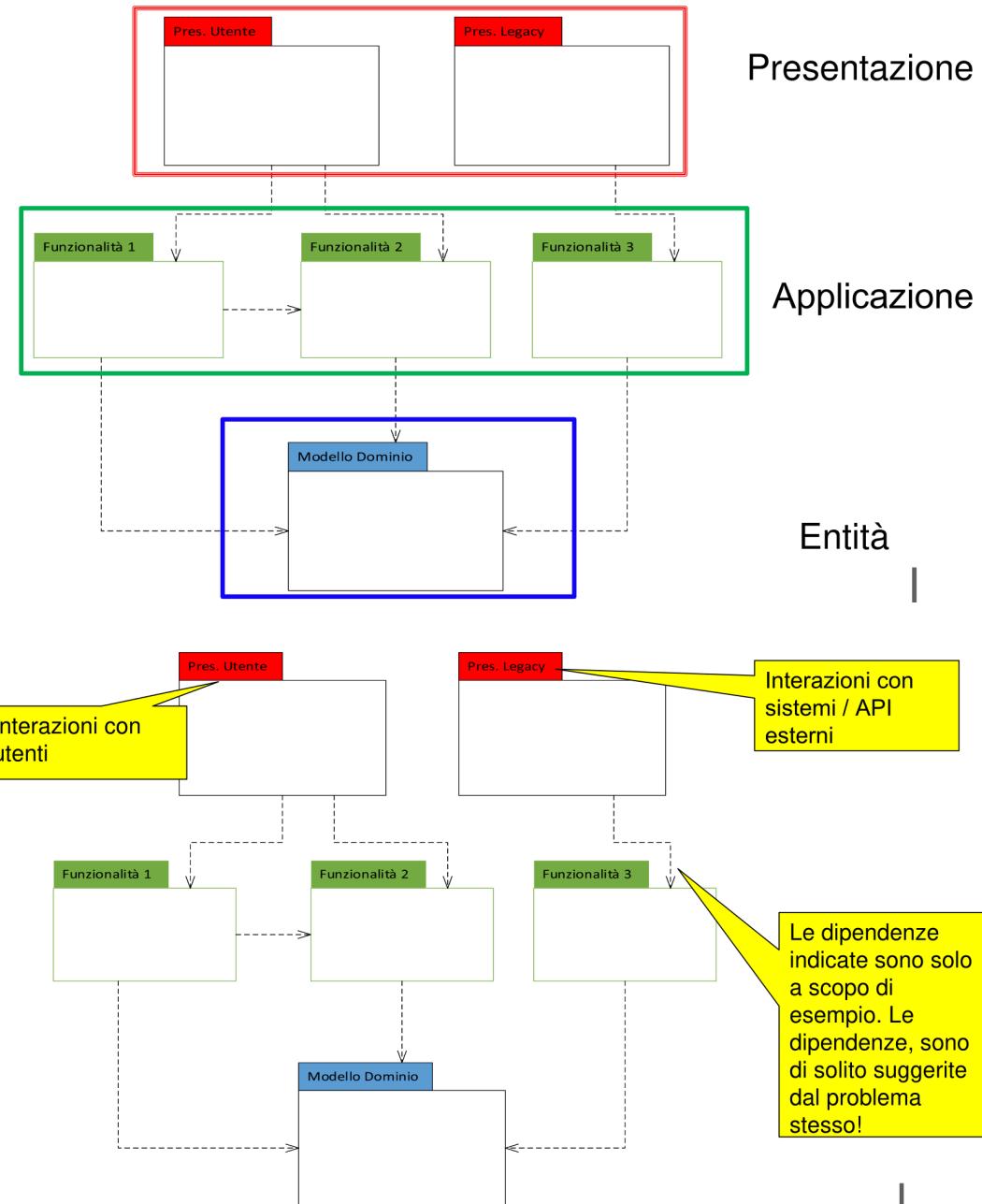
2.2.8.2 Layer

- È naturale separare la parte che realizza le entità dell'applicazione (dominio) dalla parte che realizza l'interazione con l'utente (logica di presentazione), introducendo una parte centrale di connessione (control)
- Nello specifico:
 - il livello di **presentazione** comprende le parti che realizzano l'interfaccia utente
 - Per aumentare la riusabilità delle parti, questo livello è progettato e costruito astraiendo quanto più possibile dai dettagli degli specifici dispositivi di I/O
 - il livello di **applicazione** comprende le parti che provvedono a elaborare l'informazione di ingresso, a produrre i risultati attesi e a presentare le informazioni in uscita
 - il livello delle **entità** forma il (modello del) dominio applicativo

2.2.8.3 Struttura: Package

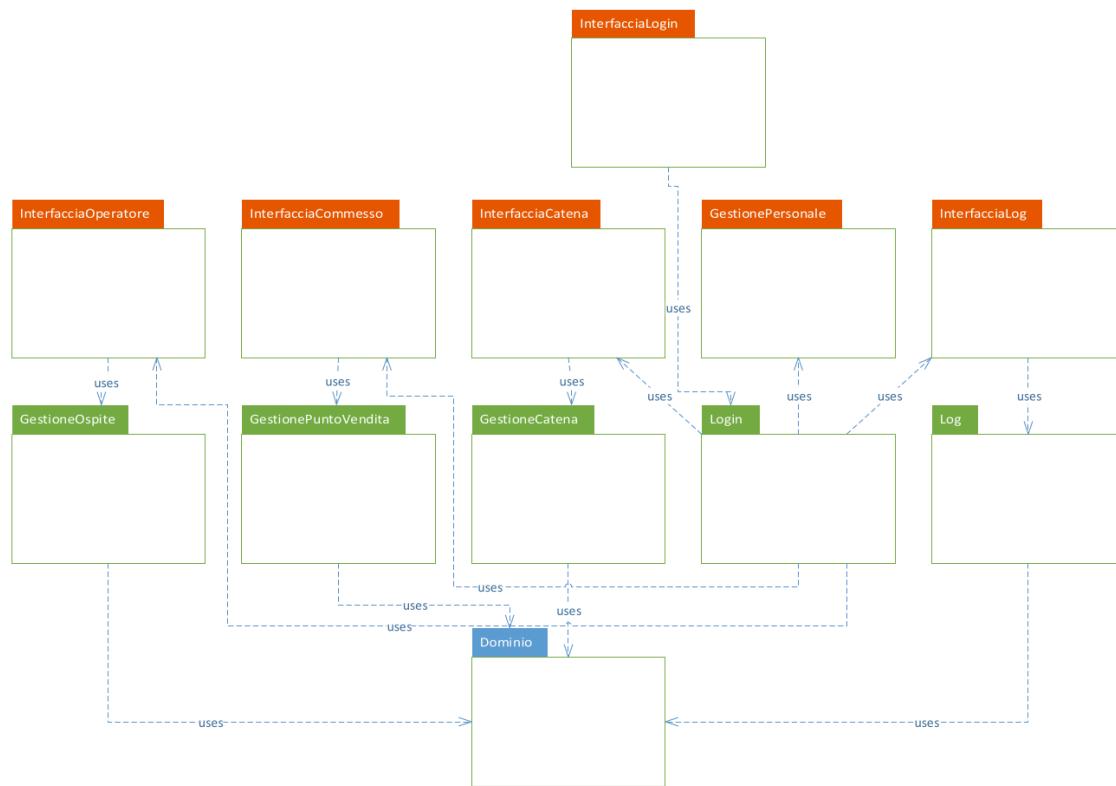
- Usando come base di partenza il lavoro di analisi svolto nelle fasi precedenti e tenendo in considerazione il pattern BCE si può iniziare la creazione del Diagramma dei Package che rappresenta la visione dal alto livello dell'Architettura Logica
- Il primo package che si può identificare è il package costituito dal **Modello del Dominio** creato nella fase precedente
 - tale Modello (se ben realizzato) costituisce la parte “**entity**” dell'architettura
- Poi è possibile creare un package per ognuna delle diverse funzionalità identificate nella Tabella delle Funzionalità (“control”); ciascuna delle righe della tabella diventa un package di controllo

- Vengono creati uno o più package per la parte di “boundary”
- Infine si identificano le *dipendenze logiche* tra i package



2.2.8.4 Esempio

Diagramma dei Package per il Villaggio Turistico

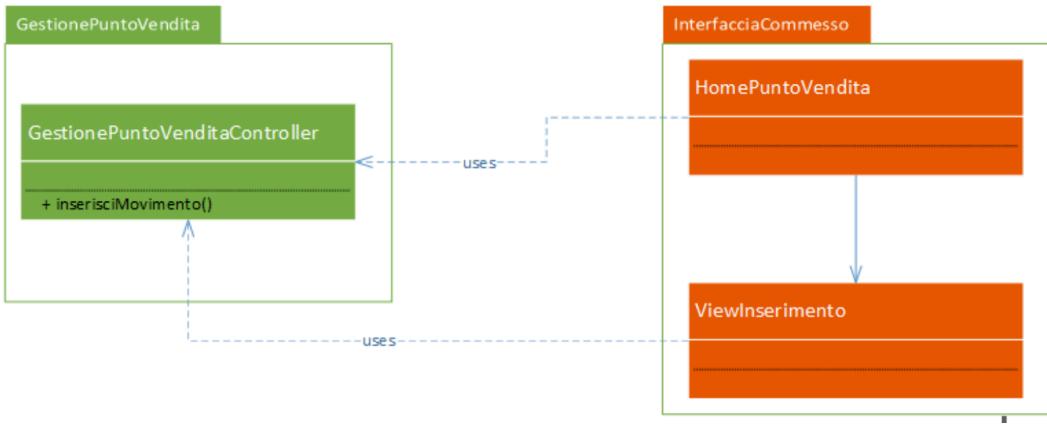


2.2.8.5 Struttura: Classi

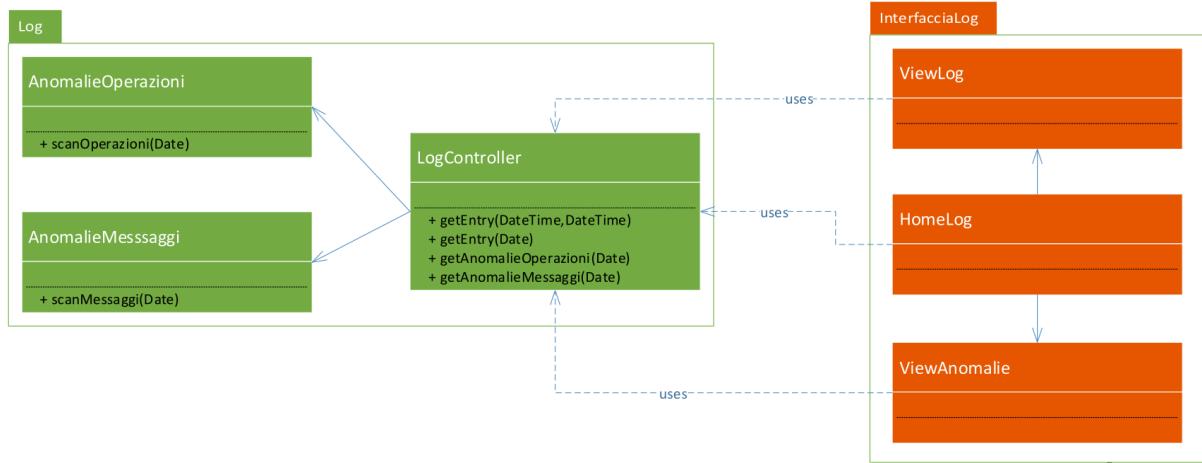
- Dopo aver stabilito la struttura di alto livello dell’Architettura Logica **si dettagliano le classi che compongono ogni package**
- Il Package del Dominio è già stato identificato nel Modello del Dominio
- Se le classi da specificare per ogni package sono poche si può realizzare un unico diagramma delle classi
- Altrimenti è possibile creare un diagramma delle classi separato per ogni package
- Attenzione a non introdurre scelte di progettazione in questa fase
- Indicare solo quelle classi che sono deducibili dal problema
 - in genere viene inserita almeno una classe che realizza le funzionalità indicate nelle specifiche
 - se nel diagramma dei package è stata indicata una funzionalità che nella fase precedente era stata poi scomposta si possono indicare le classi che realizzano le sotto-funzionalità
 - indicare le classi che rappresentano le maschere di interazione con l’utente identificate nelle fasi precedenti

2.2.8.6 Esempio

- Diagramma delle classi per il Villaggio Turistico
 - Diagramma delle classi: **InterfacciaCommesso & GestionePuntoVendita**



- Diagramma delle classi per il Villaggio Turistico
 - Diagramma delle classi: **Interfaccialog & Log**

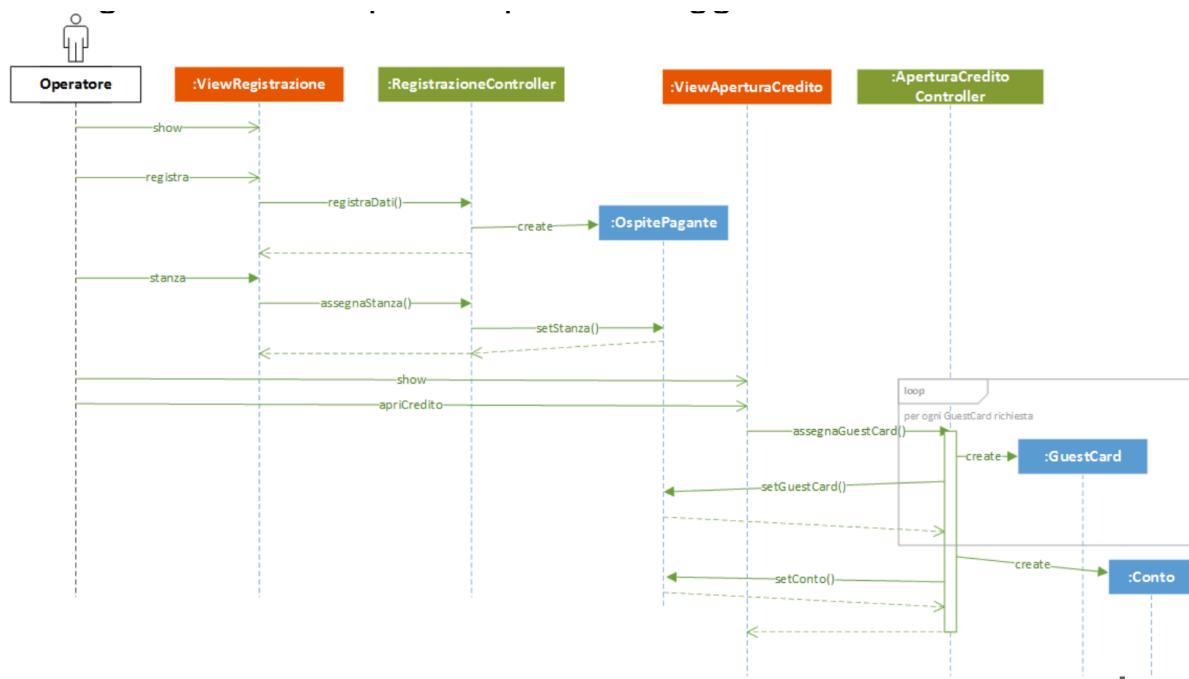


2.2.9 Architettura Logica: Interazione

- Descrivere le interazioni tra le entità identificate nella parte strutturale attraverso opportuni Diagrammi di Sequenza
- I **Diagrammi di sequenza** evidenziano
 - lo scambio di messaggi (le interazioni) tra gli oggetti
 - l'ordine in cui i messaggi vengono scambiati tra gli oggetti (sequenza di invocazioni delle operazioni)
- Non spingere la definizione dei diagrammi di sequenza sino ai minimi dettagli
- Utilizzare i diagrammi solo per descrivere il funzionamento del sistema
 - in risposta a sollecitazioni esterne
 - in fasi particolarmente significative
 - nei casi più critici
- Non serve mostrare tutti i diagrammi di interazione, ma magari fare vedere quelli più interessanti e/o complicati

2.2.9.1 Esempio

Diagramma di sequenza per il Villaggio Turistico



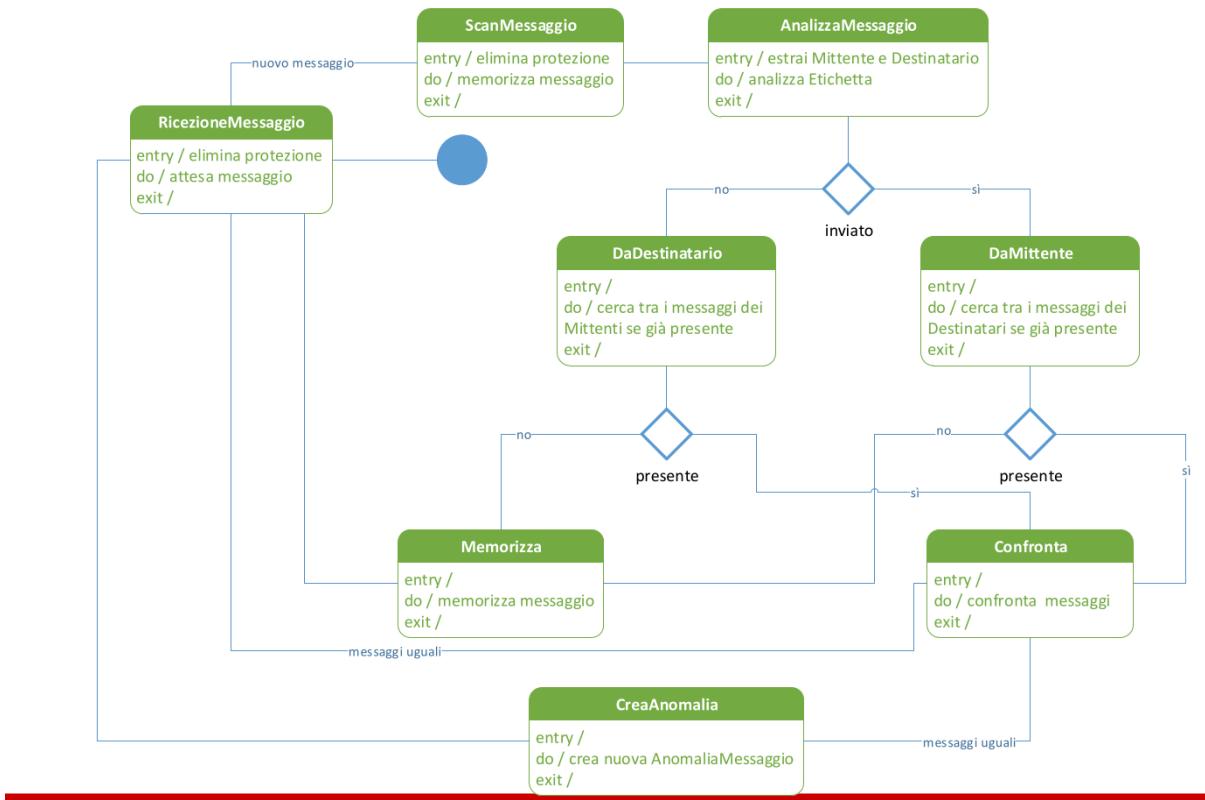
N.B. Mai fare vedere la password nel modello del dominio. L’interfaccia che gestisce il login avrà un auto-anello, quindi solo lei dovrà gestire il controllo password; come realizzare questa cosa si vedrà in progettazione.

2.2.10 Architettura Logica: Comportamento

- Descrivere il comportamento delle entità identificate nella parte strutturale attraverso opportuni Diagrammi di Stato o delle Attività
- Diagramma di Stato per mostrare come si comportano alcune entità complesse a seguito delle interazioni e degli eventi che avvengono nel sistema
- Diagramma delle Attività per dettagliare funzionamenti complessi delle entità
- Non spingere la definizione dei diagrammi sino ai minimi dettagli
- Lo stato di un oggetto è dato dal valore dei suoi attributi e delle sue associazioni
- In molti domini applicativi, esistono oggetti che, a seconda del proprio stato, rispondono in maniera diversa ai messaggi ricevuti
 - Dispositivi (spento, in attesa, operativo, guasto, ecc.)
 - Transazioni complesse (in definizione, in esecuzione, completata, fallita, ecc.)
- In questi casi, è opportuno disegnare un diagramma di stato per l’oggetto, mostrando i possibili stati e gli eventi che attivano transizioni da uno stato all’altro
- A un oggetto possono essere assegnate responsabilità che comportano un insieme di elaborazioni complesse che devono essere eseguite in un ordine particolare
- In questi casi, è opportuno disegnare un diagramma di attività per l’oggetto, mostrando le diverse elaborazioni che devono essere portate a termine e l’ordine di tali elaborazioni

2.2.10.1 Esempio

Diagramma di stato per AnomalieMessaggi



2.2.11 Definizione del Piano di Lavoro

- Dopo la creazione dell'Architettura Logica è possibile iniziare a suddividere il lavoro
- In particolare è necessario:
 - suddividere le responsabilità ai diversi membri del team di progetto e sviluppo
 - stabilire le tempistiche per la progettazione di ciascuna parte
 - stabilire i tempi di sviluppo di ciascun sotto-sistema
 - programmare i test di integrazione tra le parti
 - identificare i tempi di rilascio delle diverse versioni del prototipo
 - identificare un piano per gli sviluppi futuri

2.2.11.1 Esempio

Package	Progetto	Sviluppo
Dominio	Team progettazione + Team DB	Team sviluppo A + Team DB
Gestione Ospite	Team progettazione	Team sviluppo A
GestionePuntoVendita	Team progettazione	Team sviluppo B
GestioneCatena	Team progettazione	Team sviluppo B
Login	Team sicurezza	Team sviluppo sicurezza
Log	Team sicurezza	Team sviluppo sicurezza
InterfacciaOperatore	Team progettazione + Team grafico	Team sviluppo A + Team Grafico
InterfacciaCommesso	Team progettazione + Team grafico	Team sviluppo B + Team Grafico
InterfacciaCatena	Team progettazione + Team grafico	Team sviluppo B + Team Grafico
InterfacciaLogin	Team sicurezza+ Team grafico	Team sicurezza+ Team grafico
InterfacciaLog	Team sicurezza+ Team grafico	Team sicurezza+ Team grafico
GestionePersonale	Team progettazione + Team DB	Team sviluppo A + Team DB

2.2.12 Definizione del Piano del Collaudo

- Al termine dell’Analisi del Problema, i modelli che definiscono il dominio e l’Architettura Logica dovrebbero dare sufficienti informazioni su *cosa* le varie parti del sistema debbano fare senza specificare ancora molti dettagli del loro comportamento
- Il “*cosa fare*” di una parte dovrà comprendere anche le forme di interazione con le altre parti
- Lo scopo del *piano del collaudo* è cercare di precisare il comportamento atteso da parte di una entità prima ancora di iniziare il progetto e la realizzazione
- Focalizzando l’attenzione sulle interfacce delle entità e sulle interazioni è possibile impostare scenari in cui specificare in modo già piuttosto dettagliato la “risposta” di una parte a uno “stimolo” di un’altra parte
- Lo sforzo di definire nel modo *più preciso possibile* un piano del collaudo di un sistema prima ancora di averne iniziato la fase di progettazione viene ricompensato da
 - una *miglior comprensione* dei requisiti
 - un approfondimento nella *comprensione dei problemi*
 - una più precisa definizione dell’insieme delle funzionalità (operazioni) che ciascuna parte deve fornire alle altre per una *effettiva integrazione* nel “tutto” che costituirà il sistema da costruire
 - comprendere il *significato delle entità* e specificarne nel modo più chiaro possibile il *comportamento atteso*

2.2.12.1 Definizione Piano del Collaudo

- Un piano del collaudo va concepito e impostato da un punto di **vista logico**, cercando di individuare categorie di comportamenti e punti critici
- In molti casi tuttavia può anche risultare possibile definire in modo precoce **piani di collaudo concretamente eseguibili**, avvalendosi di strumenti del tipo JUnit/JUnit che sono ormai diffusi in tutti gli ambienti di programmazione
- Lo sforzo di definire un piano di collaudo concretamente eseguibile promuove uno sviluppo **controllato, sicuro e consapevole** del codice poiché il progettista e lo sviluppatore possono verificare subito in modo concreto la correttezza di quanto sviluppato

2.2.12.2 JUnit

- JUnit (<https://junit.org/junit5/docs/current/user-guide/>) è un framework per unit-testing per il linguaggio Java
- Dovreste conoscerlo già bene da Fondamenti T-2

2.2.12.3 NUnit

- NUnit (<http://nunit.org/>) è un framework per unit-testing per tutti i linguaggi .Net
- NUnit è inizialmente derivato da JUnit, ma è stato totalmente riscritto dalla versione 3
- Troverete la documentazione e la guida all'installazione a <https://github.com/nunit/docs/wiki/Framework-Release-Notes>

2.3 Progettazione

2.3.1 Introduzione

- **Obiettivo:** attraverso una serie di raffinamenti successivi dell'Architettura Logica arrivare ad ottenere **l'Architettura del Sistema**

Vanno considerati anche tutti gli aspetti vincolanti che sono stati trascurati nelle fasi precedenti

- Questa fase deve mirare non solo a individuare e descrivere una soluzione al problema (**what/how**), ma soprattutto a descrivere i **motivi (why)** che l'hanno determinata
- **Risultato:**
 - Architettura del Sistema
 - Schema Persistenza
 - Piano finale del Collaudo
 - Indicazioni per il Deployment



1. Progettazione Architetturale

Obiettivo: definire l'Architettura del Sistema tenendo conto di tutti i vincoli e delle forze in gioco

2. Progettazione di Dettaglio

Obiettivo: progettare nel dettaglio ogni aspetto del Sistema

3. Progettazione della Persistenza

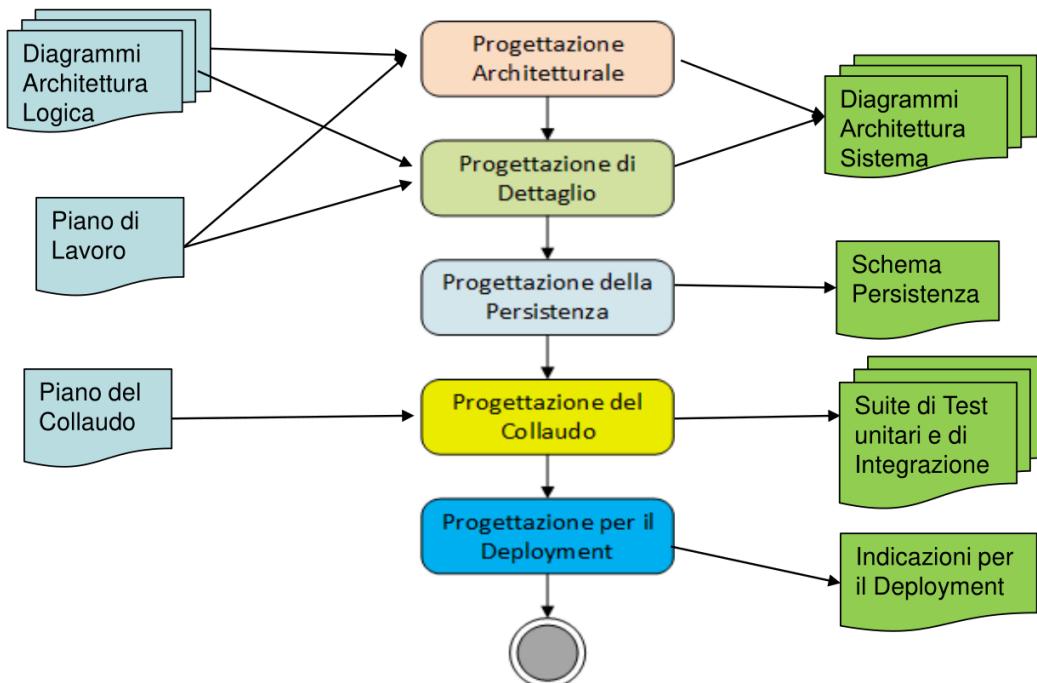
Obiettivo: progettare i meccanismi per la persistenza dei dati

4. Progettazione del Collaudo

Obiettivo: definire in modo chiaro e preciso come il sistema dovrà essere collaudato una volta terminata l'implementazione

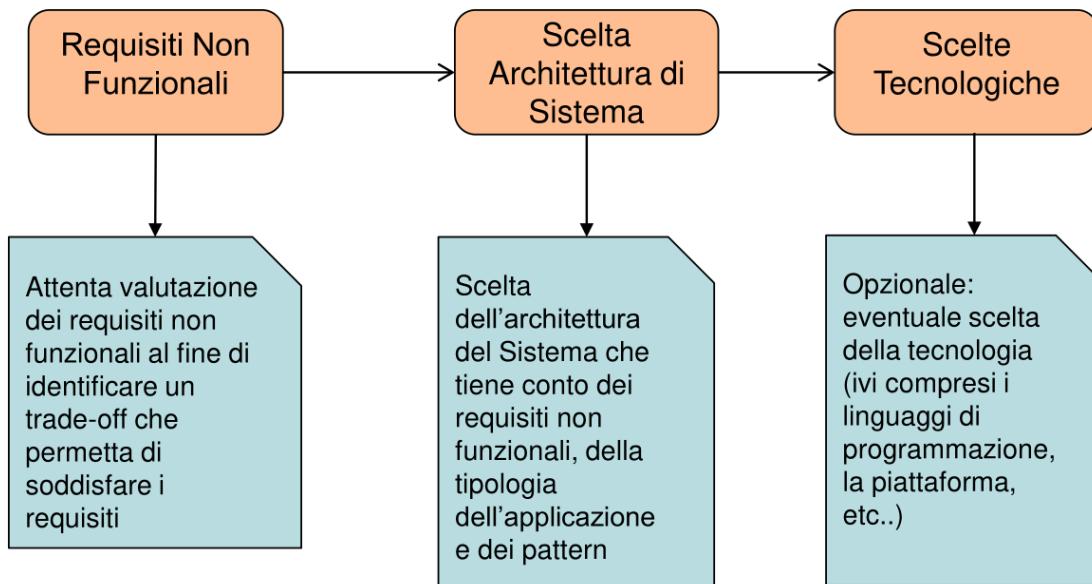
5. Progettazione per il Deployment

Obiettivo: progettare il sistema in modo da rendere semplice il deployment sulle macchine e per garantire la sicurezza



2.3.2 Progettazione Architetturale

- Nella Progettazione Architetturale gli ingegneri devono prendere delle decisioni che influenzano profondamente il sistema
- Basandosi sulle proprie esperienze e conoscenze devono rispondere ad alcune domande fondamentali:
 - C'è un'architettura applicativa generica che può essere utilizzata come modello per il sistema che sto progettando?
 - Come sarà distribuito il sistema tra più processori?
 - Quale stile o quali stili sono adatti al sistema?
 - Quale sarà l'approccio fondamentale utilizzato per strutturare il sistema?
 - Come saranno scomposte in moduli le unità strutturali del sistema?
 - Quale strategia sarà usata per controllare l'operato delle unità del sistema?



2.3.2.1 Requisiti Non Funzionali

- L’architettura del sistema influenza
 - le **prestazioni**
 - la **robustezza**
 - la **distribuibilità**
 - la **manutenibilità**
- di un sistema
- La struttura dell’architettura tipicamente è condizionata
 - dalla **tipologia di applicazione** che si vuole realizzare
 - dai **requisiti non funzionali**
- Se le **prestazioni** sono un requisito critico l’architettura dovrebbe essere progettata
 - localizzando le operazioni critiche all’interno di un piccolo numero di componenti
 - minimizzando le comunicazioni possibile tra essi
- Questo porta a dover definire componenti “**grandi**” per ridurre la comunicazione
- Se la **protezione dei dati (security)** è un requisito critico l’architettura dovrebbe essere progettata
 - con una struttura “stratificata”
 - collocando le risorse più critiche nello strato più interno e protetto
- Questo porta a dover definire una struttura con un alto livello di convalida di protezione a ogni strato
- **NB:** Quando si valuta l’aspetto della protezione dei dati tenere conto di tutte le indicazioni che sono emerse nella parte della Security Engineering
- Se la sicurezza (safety) è un requisito critico l’architettura dovrebbe essere progettata
 - in modo tale che le operazioni relative siano tutte collocate in un singolo componente o in un piccolo insieme di componenti

- riduzione dei costi e dei problemi di convalida della sicurezza, possibilità di poter fornire sistemi di protezione correlati
- Questo porta a dover definire componenti “grandi” per localizzare le operazioni
- Se la **disponibilità** è un requisito critico l’architettura dovrebbe essere progettata
 - per comprendere componenti ridondanti
 - in modo che sia possibile sostituirli e aggiornarli senza fermare il sistema
- Questo porta a dover sviluppare un numero maggiore di componenti rispetto a quelli strettamente necessari
- Se la **manutenibilità** è un requisito critico l’architettura dovrebbe essere progettata
 - usando componenti piccoli, atomici, autonomi
 - che possano essere modificati velocemente
 - i produttori di informazione dovrebbero essere separati dai consumatori e le strutture dati condivise dovrebbero essere evitate
- Questo porta a dover sviluppare componenti di piccole dimensioni
- Ci sono dei conflitti potenziali tra alcune di queste architetture così come abbiamo visto sussistono conflitti tra i requisiti non funzionali
- Esempio: usare componenti “grossi” migliora le prestazioni ma peggiora la manutenibilità e viceversa
- Se sono entrambi requisiti critici occorre trovare un compromesso

2.3.2.2 Esempio

Nell’Analisi del Problema (Tabella Vincoli) sono emersi tre requisiti non funzionali che impongono dei vincoli al sistema:

- Tempo di risposta
- Usabilità
- Sicurezza

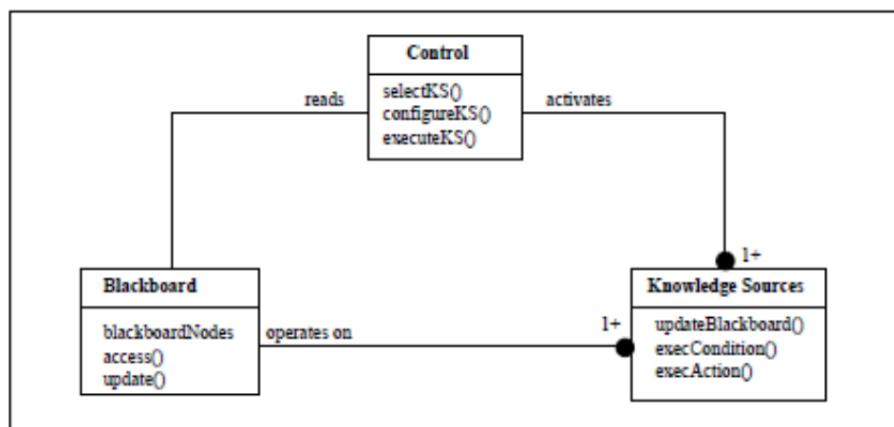
Nello specifico caso in esame, Usabilità e Sicurezza hanno pochi conflitti a parte l’eventuale richiesta di un ulteriore login se per caso scade la sessione di lavoro. L’Usabilità impatta molto di più la struttura delle interfacce che andranno progettate in modo tale da mantenere nelle stesse View le informazioni necessarie alle funzionalità richieste. Diversa la questione che riguarda Tempo di risposta e Sicurezza, aggiungere strati (layer) e meccanismi di cifratura per migliorare la sicurezza ovviamente porta ad un peggioramento delle prestazioni del sistema, occorre quindi trovare un bilanciamento tra i due aspetti. Considerando la tipologia di sistema che deve essere sviluppato, si ritiene maggiormente critico l’aspetto di sicurezza dei dati in quanto la “Tabella Valutazione Beni” mette in luce che nel caso di attacchi al sistema andati a buon fine si rischia un’esposizione molto alta con perdite finanziarie e di immagine. Inoltre, gli utenti principali di tale sistema sono operatori umani che spesso non sono in grado di percepire se il Sistema impiega qualche frazione di secondo in più o in meno nella risposta, non si hanno vincoli real-time da soddisfare.

2.3.2.3 Scelta Architettura

- La scelta dell'Architettura del Sistema deve basarsi su:
 - Architettura Logica definita in fase di Analisi del Problema
 - Trade-off requisiti non funzionali
 - Tipologia di applicazione che si intende sviluppare
 - Adozione di Pattern Architetturali
 - Blackboard
 - MVC/BCE
 - Layers
 - Client/Server
 - Broker
 - Pipe & Filters
 - ...

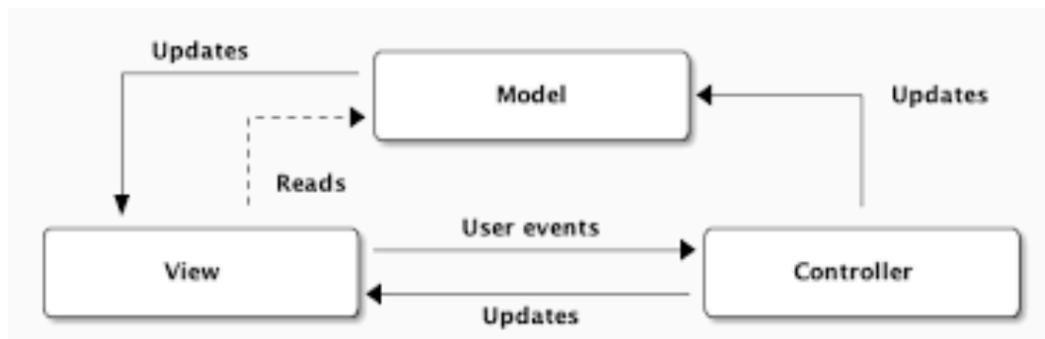
2.3.2.3.1 Blackboard

- Il pattern Blackboard aiuta a strutturare quelle applicazioni in cui vengono applicate strategie di soluzione non deterministiche (tipici problemi di intelligenza artificiale)
- I diversi sotto-sistemi condividono la stesse conoscenze attraverso la Blackboard al fine di costruire una soluzione approssimata o parziale



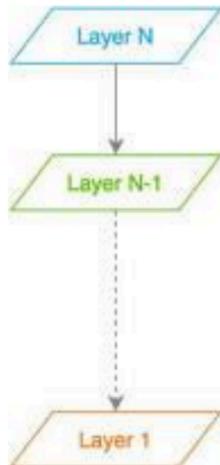
2.3.2.3.2 MVC

- Il pattern MVC divide le applicazioni in tre distinte parti:
 - Il model che gestisce i dati
 - Il controller che manipola i dati
 - La view che mostra i dati



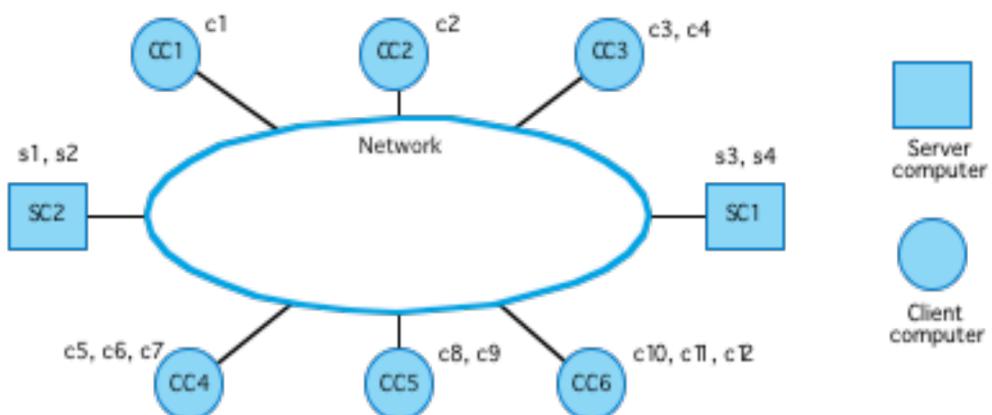
2.3.2.3.3 Layer

- Il pattern Layer aiuta a strutturare quelle applicazioni che possono essere scomposte in gruppi di sotto-attività in cui ciascun gruppo si trova a un ben definito livello di astrazione

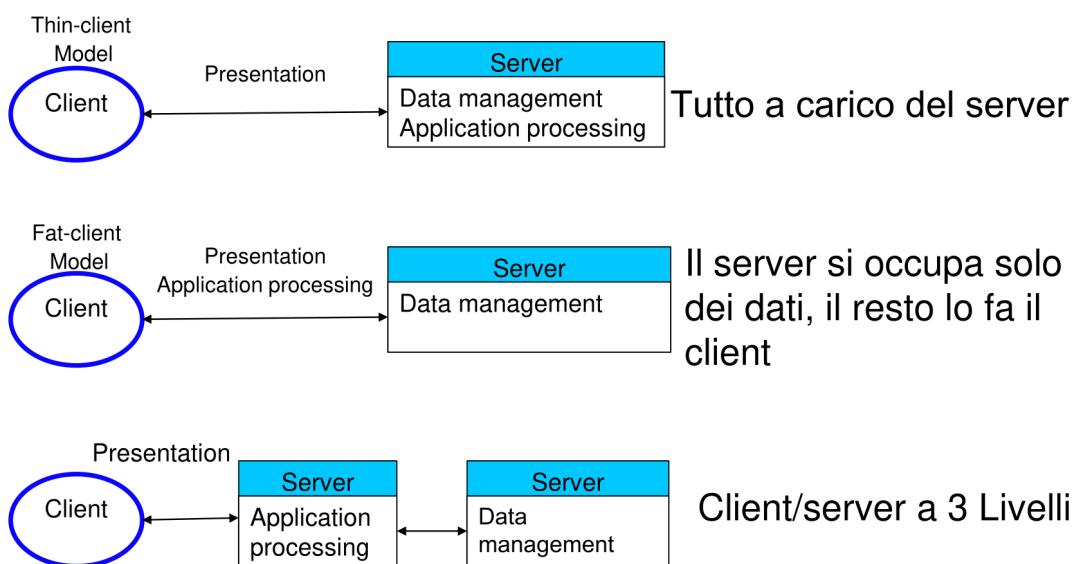


2.3.2.3.4 Client/Server

- Il pattern client/server aiuta a strutturare un'applicazione come un insieme di servizi forniti da uno o più server e un insieme di client che utilizza tali servizi

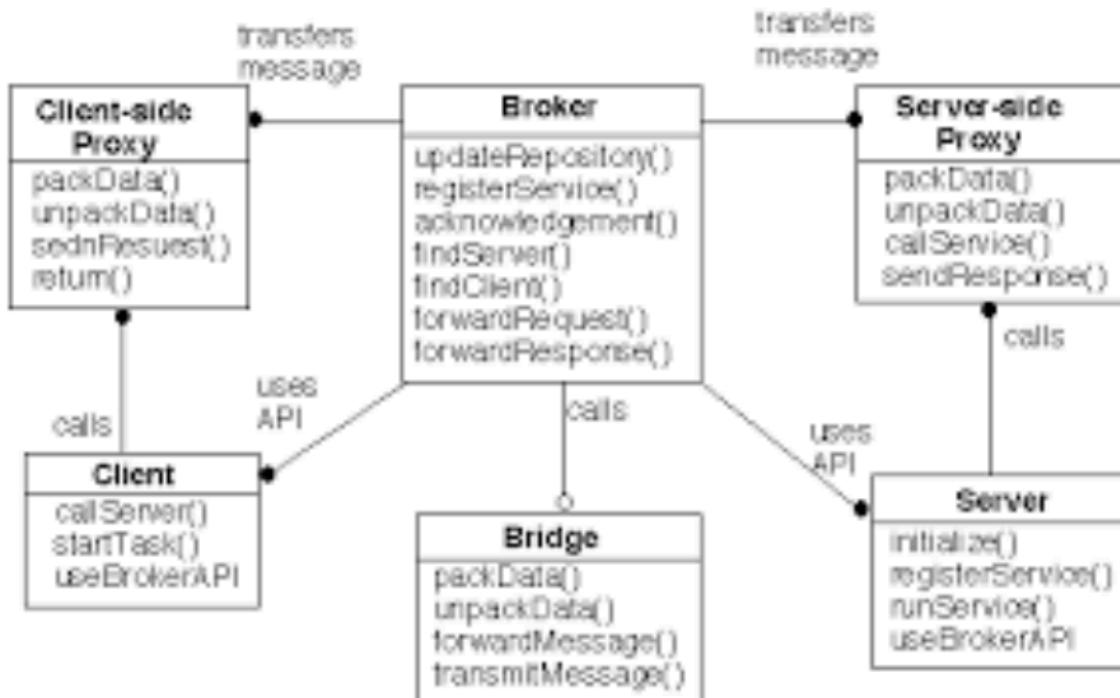


Tre diverse possibilità:



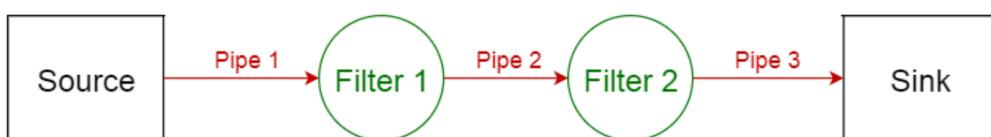
2.3.2.3.5 Broker

- Il pattern Broker può essere usato per strutturare sistemi distribuiti con un disaccoppiamento tra i diversi sotto-sistemi che comunicano tra loro attraverso remote server invocation
 - Il Broker è responsabile della coordinazione delle comunicazioni, come inoltro richieste, invio risposte ed eccezioni



2.3.2.3.6 Pipe & Filters

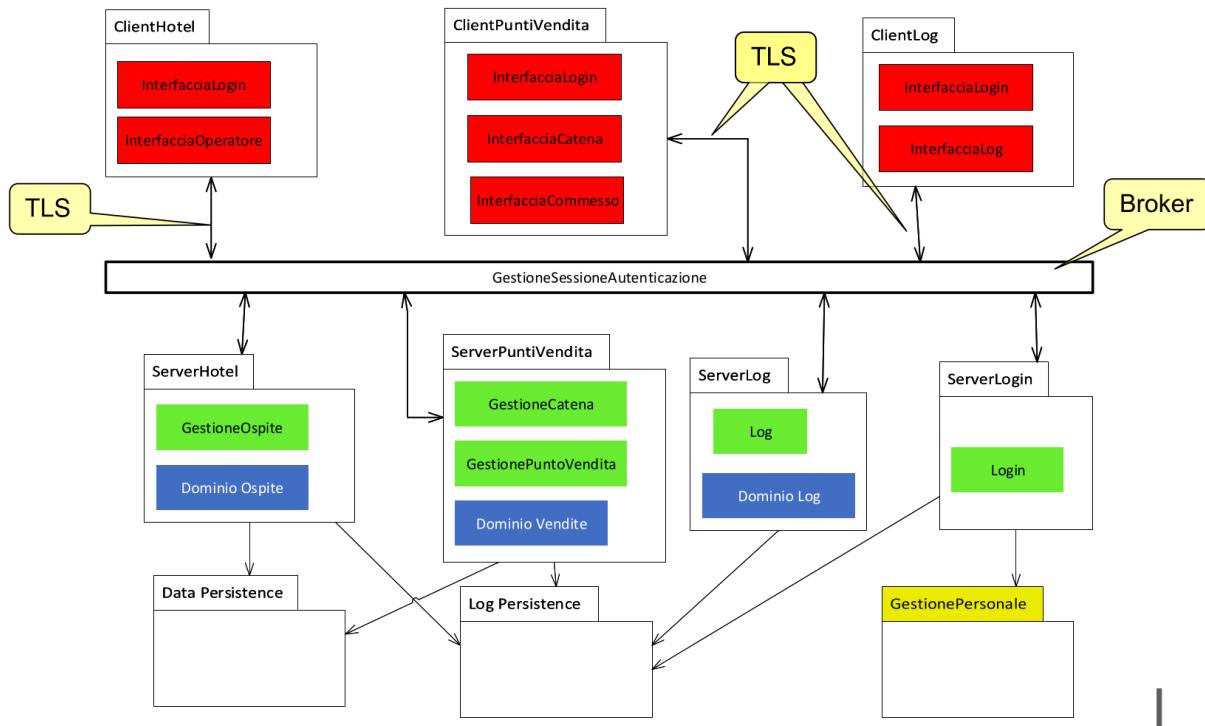
- Il pattern Pipe & Filters aiuta a strutturare quelle applicazioni che processano flussidi dati
 - Ogni passo del processo è incapsulato in un apposito filtro e i dati attraversano una pipe di filtri
 - Variando l'ordine dei filtri si possono ottenere diversi tipi di sistemi



2.3.2.3.7 Conclusioni

- Come non esiste un processo di sviluppo ideale, non esiste un'Architettura ideale sempre utilizzabile
 - Talvolta è necessario usare stili architetturali diversi per parti diverse del sistema al fine di soddisfare tutti i vincoli imposti dai requisiti
 - L'adozione dei pattern architetturali può aiutare a trovare il giusto compromesso tra tutte le forze in gioco

2.3.2.3.8 Esempio: Villaggio Turistico



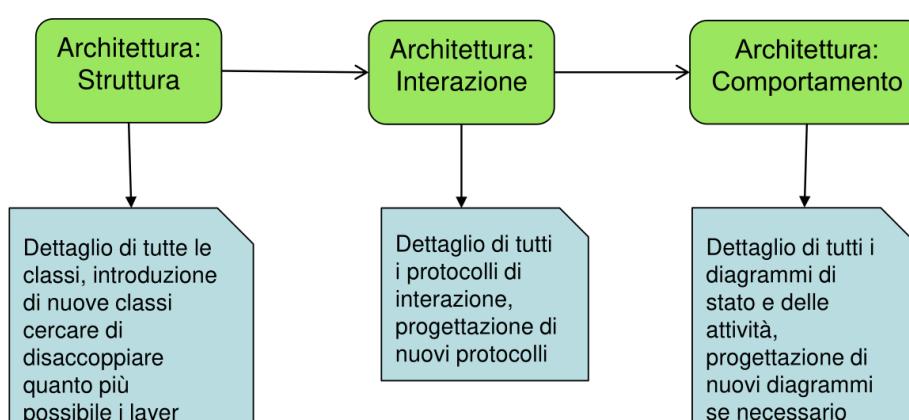
2.3.2.4 Scelte Tecnologiche

- L'uso di una specifica tecnologia (intesa anche come linguaggio di programmazione, piattaforma, strumento, etc.) non è sempre neutro
- In taluni casi potrebbe risultare vantaggioso scegliere le tecnologie già in fase di progettazione legando così il progetto alla specifica tecnologia
- Nel caso si decida di scegliere la tecnologia in fase di progettazione, va specificato chiaramente e va fatta un'analisi costi/benefici
- Vanno attentamente studiate le parti della tecnologia adottata in modo che sia poi possibile inserirle nei diagrammi di progettazione

2.3.3 Progettazione di dettaglio

- La Progettazione di Dettaglio definisce il dettaglio dell'Architettura del Sistema nelle sue tre viste:
 - Struttura
 - Interazione
 - Comportamento
- Quindi non più architettura logica, ma architettura di sistema
- Per realizzare un sistema funzionante, occorre considerare GUI, DB, Framework, librerie, componenti, modifiche al modello per avere **software estensibile e modulare...**
- È compito della Progettazione di Dettaglio **identificare e definire altre classi** in accordo alla specifica architettura scelta, siamo arrivati al *come*
- Durante la Progettazione di Dettaglio, i modelli prodotti nell'Analisi devono essere **estesi** al fine di progettare i quattro layer principali che compongono il sistema
 - **Application Logic** - logica dell'applicazione e controllo degli altri componenti

- ▶ **Presentation logic** - gestione dell'interazione con l'utente a livello logico nuovi oggetti: finestre, menu, button, toolbar , ...
- ▶ **Data logic** - gestione dei dati che il sistema deve manipolare
- ▶ **Middleware** - gestione dell'interazione con i sistemi esterni, con la rete e tra i sotto-sistemi
- Durante la Progettazione di Dettaglio, i modelli di Analisi devono essere **modificati** al fine di:
 - ▶ definire in dettaglio le classi e delle loro relazioni
 - ▶ supportare **caratteristiche specifiche** per comunicazioni,
 - ▶ diagnostica, protezione dei dati,...
 - ▶ **riuso** di classi e/o componenti disponibili
 - ▶ miglioramento delle **prestazioni**
 - ▶ supporto alla **portabilità**
 - ▶ ...
- **Massima indipendenza possibile** da
 - ▶ Linguaggio (e ambiente) di programmazione
 - ▶ DBMS
 - ▶ Sistema Operativo
 - ▶ Hardware
- Le caratteristiche specifiche del contesto utilizzato devono essere tenute in conto solo se
 - ▶ **sono vincolanti** (requisiti non funzionali)
 - ▶ si è **esplicitamente scelto** di legarsi a una tecnologia nella progettazione architettonica, ad esempio una specifica tecnologia utilizzabile solo da un linguaggio di programmazione

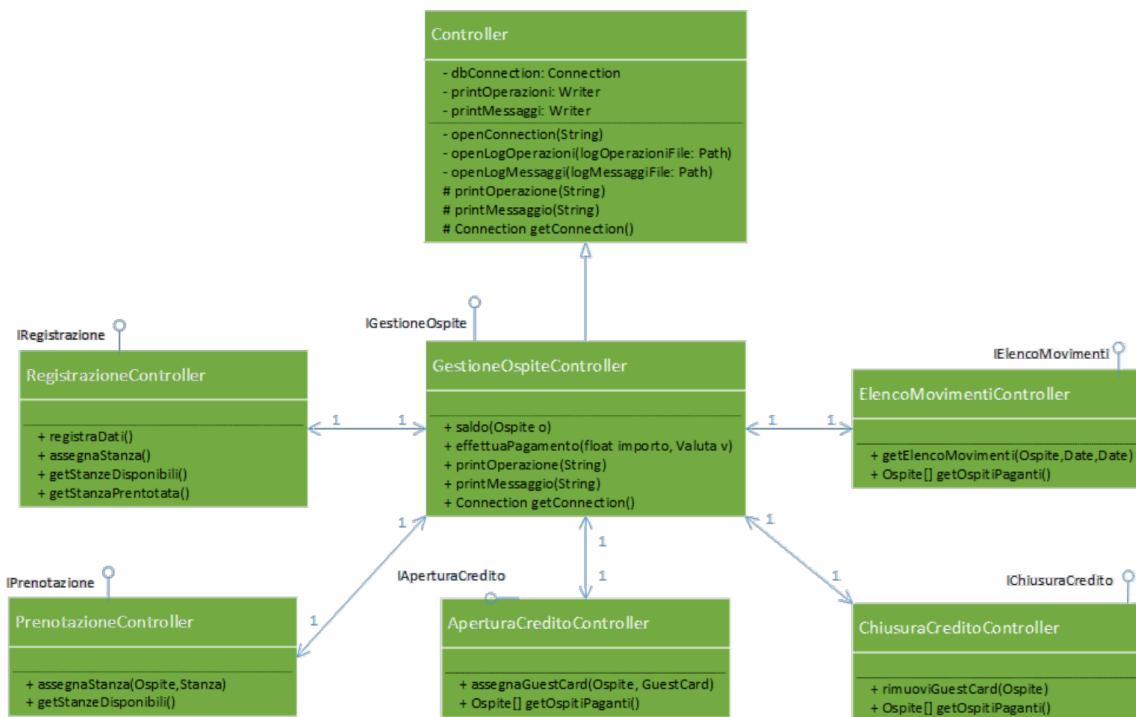
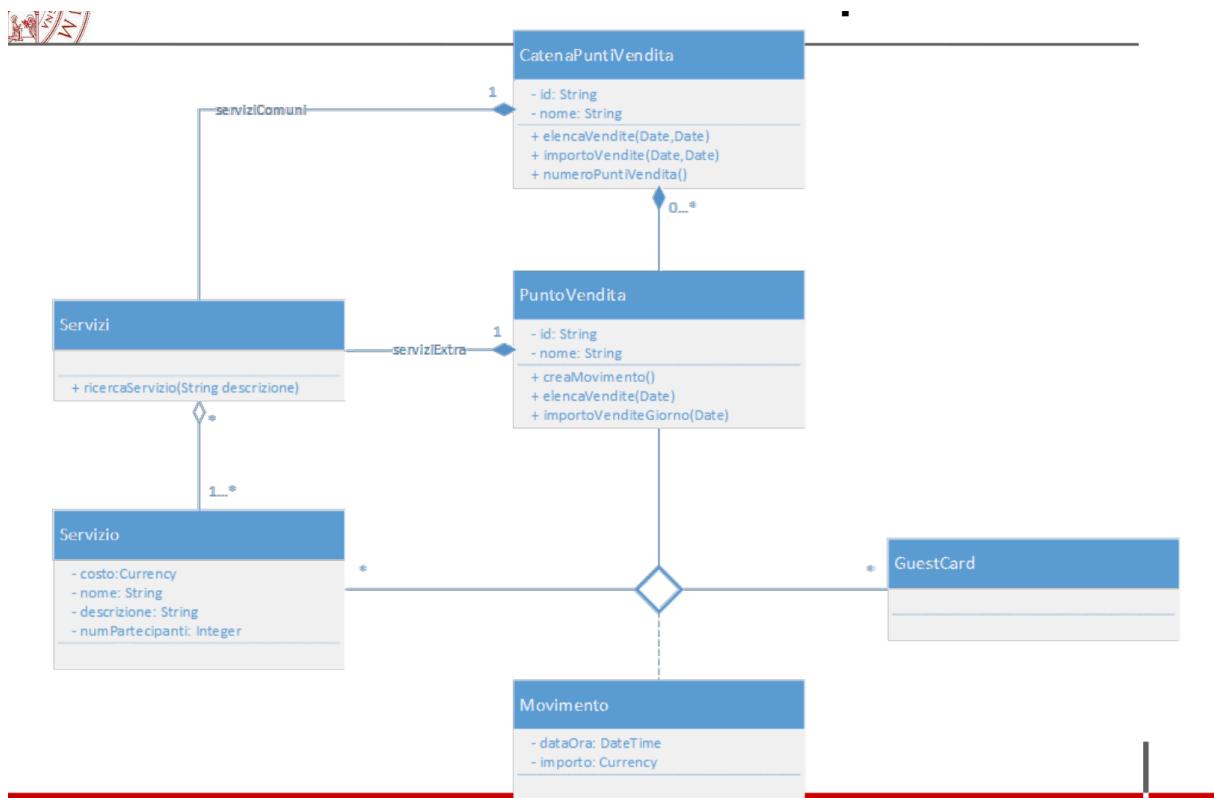


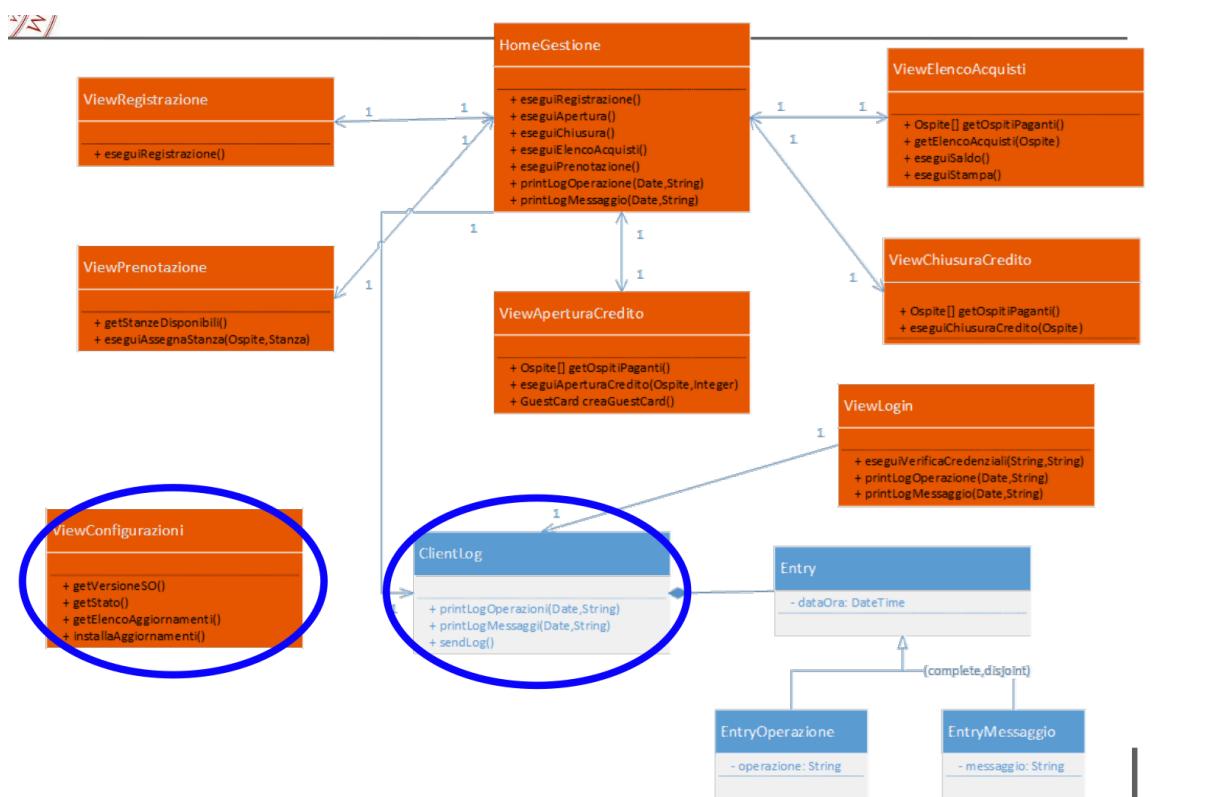
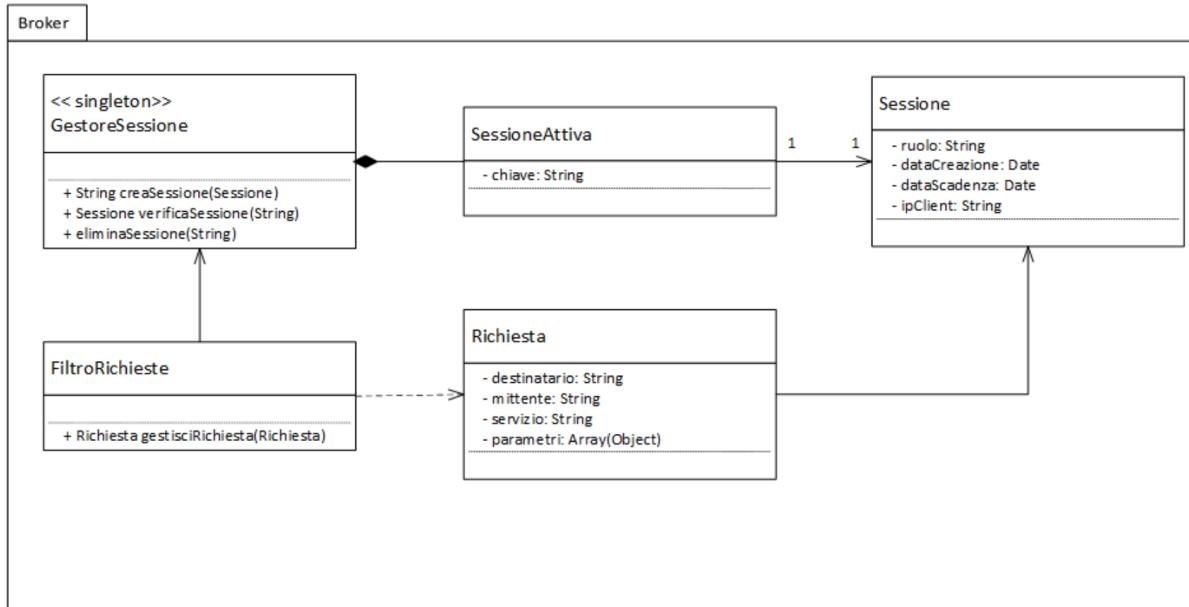
2.3.3.1 Architettura: Struttura

- Durante la Progettazione di Dettaglio della parte di Struttura è necessario definire
 - ▶ **tipi di dato** che non sono stati definiti in precedenza

- ▶ **navigabilità delle associazioni tra classi**
- ▶ **strutture dati** necessarie per l'effettiva implementazione del sistema
- ▶ **operazioni** che non erano emerse durante la fase di Analisi del Problema
- ▶ eventuali **nuove classi** necessarie per il corretto funzionamento del sistema
- Attenzione alla presenza dei “Sistemi Esterni” individuati in fase di Analisi del Problema
- Se nella tabella “Tabella dei Sistemi Esterni” era stato individuato un problema nel “Livello di Protezione” e il Sistema Esterno non risulta avere il livello di sicurezza minimo richiesto occorre applicare il **pattern Adapter**
 - ▶ si ingloba (wrappa) il Sistema Esterno in una nostra struttura
 - ▶ si progetta la struttura in modo tale che soddisfi i livelli minimi di sicurezza richiesti
- Attenzione se si è deciso di vincolarsi a una specifica tecnologia
- Va condotta una **attenta analisi** e **valutazione del livello di protezione offerto** dalla tecnologia scelta
- Se tale livello non risulta essere quello minimo richiesto dall'applicazione occorre progettare specifiche parti del sistema per prevenire i buchi di sicurezza legati alla specifica tecnologia
- Ove possibile cercare di applicare il pattern Adapter
- Applicazione dei **design pattern** al fine di realizzare **software di qualità** facilmente estensibile e modulare
- Applicazione dei **principi di progettazione** con particolare attenzione al **“Dependency Inversion Principle”**
- Disaccoppiare i layer del sistema porta molti vantaggi
 - ▶ possibile cambiare implementazione di parti del sistema senza che la modifica si ripercuota sulla restante parte: **design for change**
 - ▶ possibile cambiare l'aspetto grafico anche variando la tecnologia realizzativa senza dover modificare l'application logic
 - ▶ facile inserire nuove funzionalità con impatto minimo sul sistema

2.3.3.2 Struttura: Esempio





Sistema di Gestione Villaggio Turistico

username	<input type="text"/>
password	<input type="password"/>
Accedi	

Sistema di Gestione Ospiti

[Registra](#) [Apertura Credito](#) [Chiusura Credito](#)

[Elenco Acquisti](#) [Prenota](#)

Registrazione

Nome	<input type="text"/>
Cognome	<input type="text"/>
Indirizzo	<input type="text"/>
Telefono	<input type="text"/>
Data nascita	<input type="text"/>
Estremi Documento	<input type="text"/>
Data Inizio	<input type="text"/> ▼
Data Fine	<input type="text"/> ▼
Stanza	<input type="text"/> ▼
Carta Credito	<input type="text"/>
Pagante	<input type="checkbox"/>

[Registra](#) [Annulla](#)

Apertura Credito

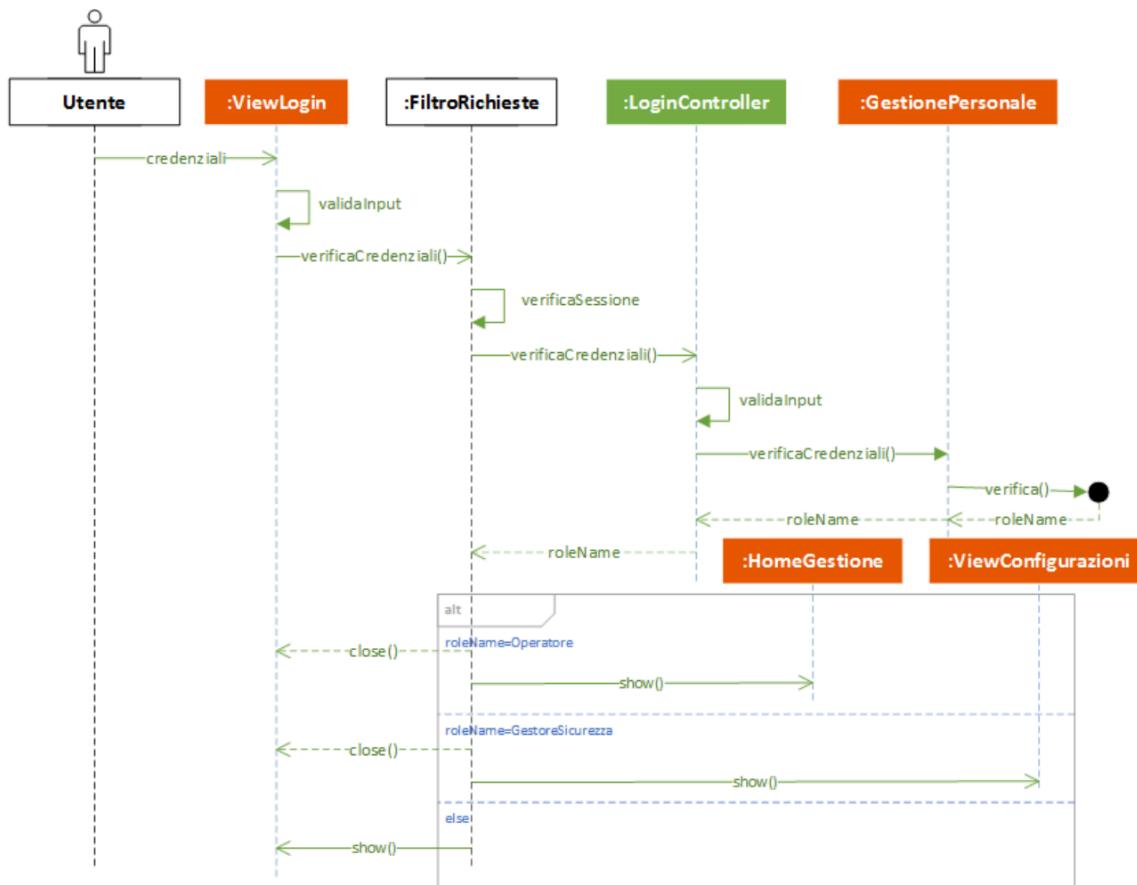
Ospite Pagante	<input type="text"/> ▼
Num GuestCard	<input type="text"/>

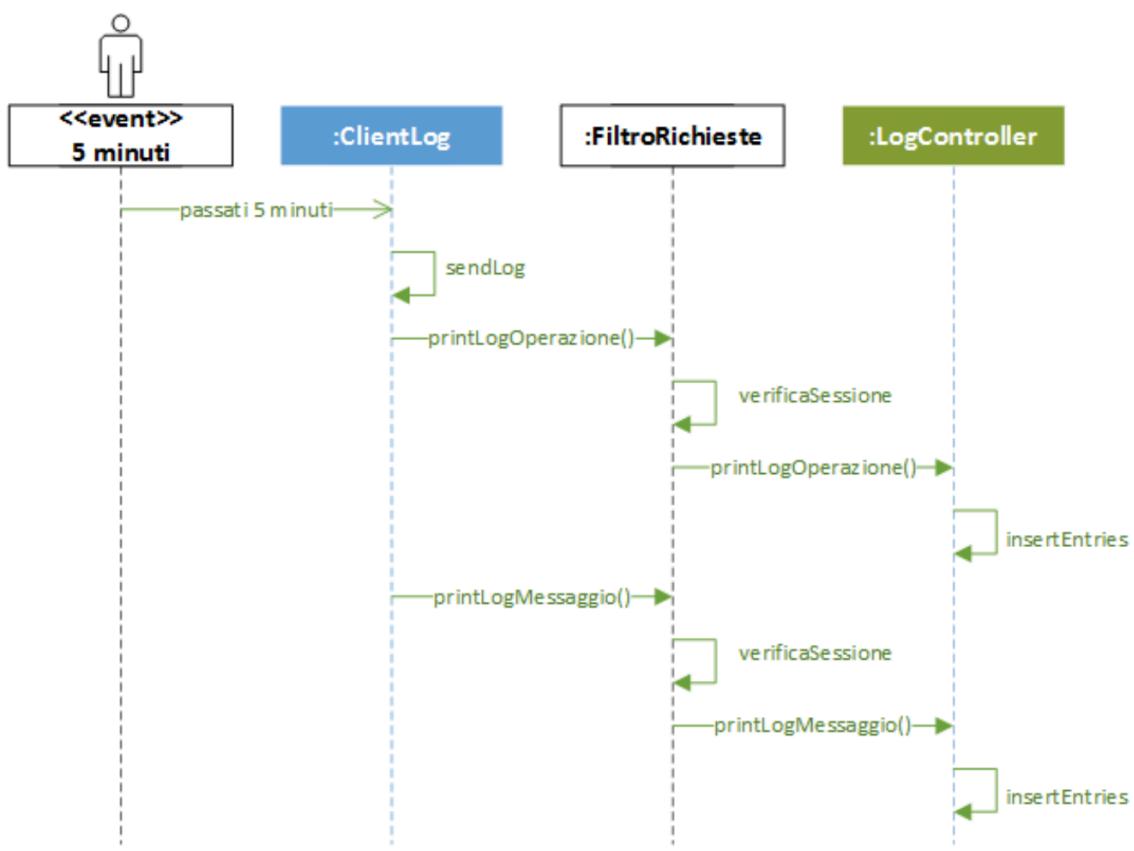
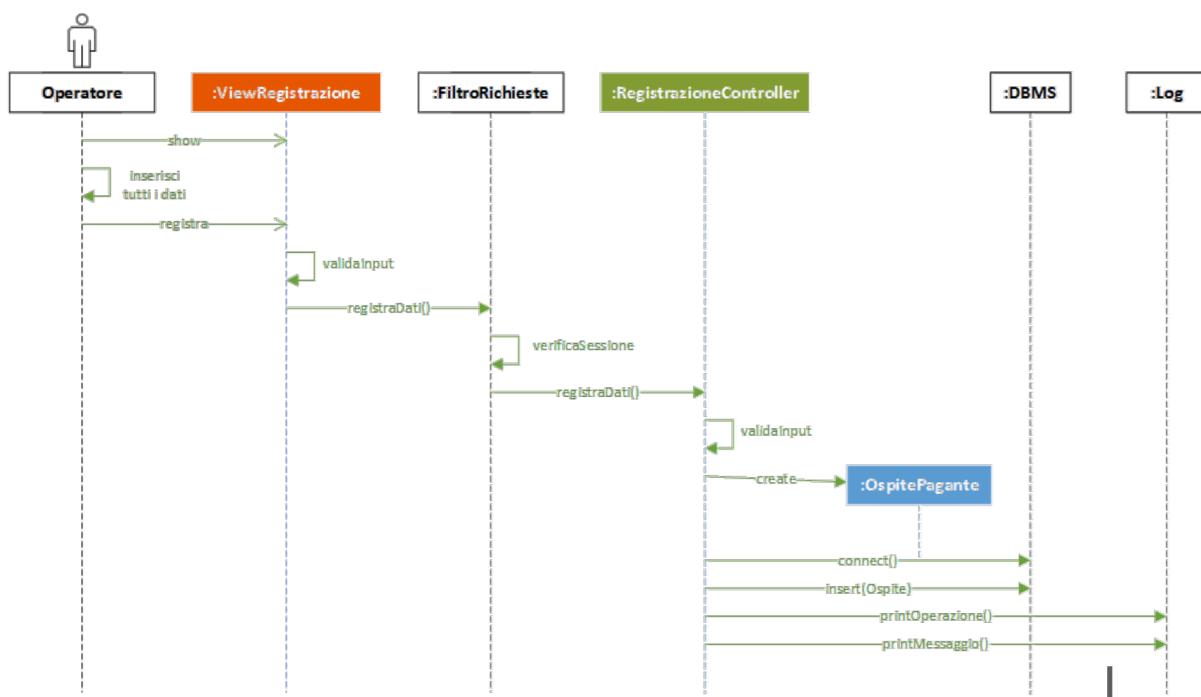
[Apri](#) [Annulla](#)

2.3.3.3 Architettura: Interazione

- Durante la Progettazione di Dettaglio della parte di Interazione è necessario
 - ridefinire i protocolli di interazione emersi in fase di Analisi dettagliandoli tenendo conto delle nuove entità emerse in progettazione, quindi definiamo il *come*
 - progettare accuratamente i protocolli di interazione verso i sistemi esterni
 - definire nuovi protocolli di interazione tra le classi che sono state introdotte nella progettazione

2.3.3.3.1 Interazione: Esempio





2.3.4 Architettura: Comportamento

- Durante la progettazione di dettaglio della parte di Comportamento è necessario
 - **definire gli algoritmi** che implementano le operazioni complesse/complicate in modo chiaro e preciso avvalendosi eventualmente di diagrammi delle attività
 - **dettagliare** i diagrammi di stato/attività già definiti nella fase precedente
 - eventualmente **aggiungere diagrammi** di stato/attività per le nuove entità emerse in questa fase

2.3.5 Progettazione della persistenza

- La persistenza dei dati è un fattore cruciale nello sviluppo di un sistema

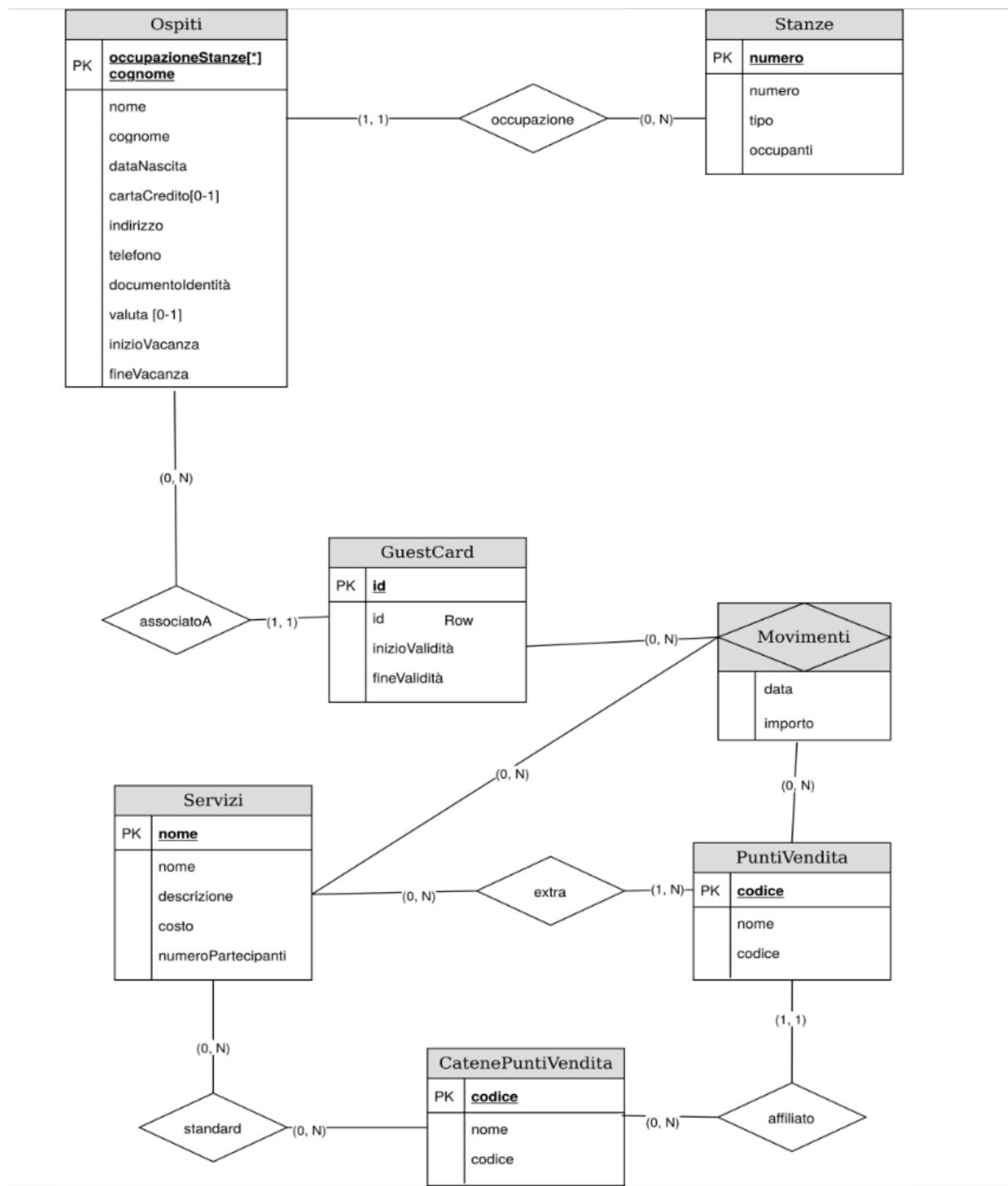
- Il progettista dopo un'attenta valutazione di
 - ▶ vincoli imposti dai requisiti funzionali (tempi di risposta, requisiti di protezione e privacy, ...)
 - ▶ tipologia di accesso accesso ai dati (lettura, scrittura, ricerche)
 - ▶ frequenza di accesso ai dati (quanto spesso devo accedere ai dati?)
 - ▶ criticità e consistenza dei dati (quanto spesso cambiano i dati? quali sono i costi di eventuale “perdite” nelle modifiche dei dati?)
- dovrà scegliere la tecnica migliore di persistenza
- Per ogni sistema va valutato attentamente quale strategia dà il miglior bilanciamento tra i vincoli e le forze in gioco nel sistema
 - Non è sempre detto che l’adozione di un (R)DBMS sia la risposta corretta
 - Per esempio se dobbiamo memorizzare dei log la strategia migliore è quella delle scrittura su file:
 - ▶ la maggior parte delle funzionalità “scrivono” solamente una o più righe nel log e l’accesso deve essere molto veloce: il log non deve pesare troppo nei tempi di risposta nel sistema
 - ▶ solo gli strumenti di analisi accedono in lettura al loge solitamente occorre analizzare ogni singola riga nel corretto ordine temporale, non c’è bisogno di fare ricerche

2.3.5.1 Quando usare un DBMS

- In generale possiamo affermare che quando si ha a che fare con:
 - ▶ gestionali che trattano un numero considerevole di dati anche di natura eterogenea
 - ▶ dati che cambiano molto spesso e devono essere costantemente aggiornati
 - ▶ la “perdita” di modifiche può essere un problema
 - ▶ necessità di ripristino di versioni precedenti a seguito di un malfunzionamento
- la scelta consigliata è quella di avvalersi di un DBMS**
-

- L’output di questa fase può essere rappresentato da:
 - ▶ lo schema E-R del DB che dovrà supportare l’applicazione
 - ▶ il formato del/i file che dovranno essere scritti/letti dall’applicazione
- Sarebbe bene che sia nel caso di DB che di file ci fosse una **piccola analisi del rischio** per capire se
 - ▶ il DB è protetto in modo adeguato
 - ▶ il/i file necessitano di meccanismi di protezione
- Il punto di partenza di tale analisi sono **i livelli di protezione e privacy richiesti per i diversi dati** che saranno memorizzati

2.3.5.2 Esempio: DB Villaggio Turistico



2.3.5.3 Esempio: log Villaggio Turistico

- Formato file per Log delle operazioni
 - DataOra operazione esecutore
- Formato file per Log dei messaggi
 - DataOra messaggio protetto invio/ricezione autore

2.3.6 Progettazione del collaudo

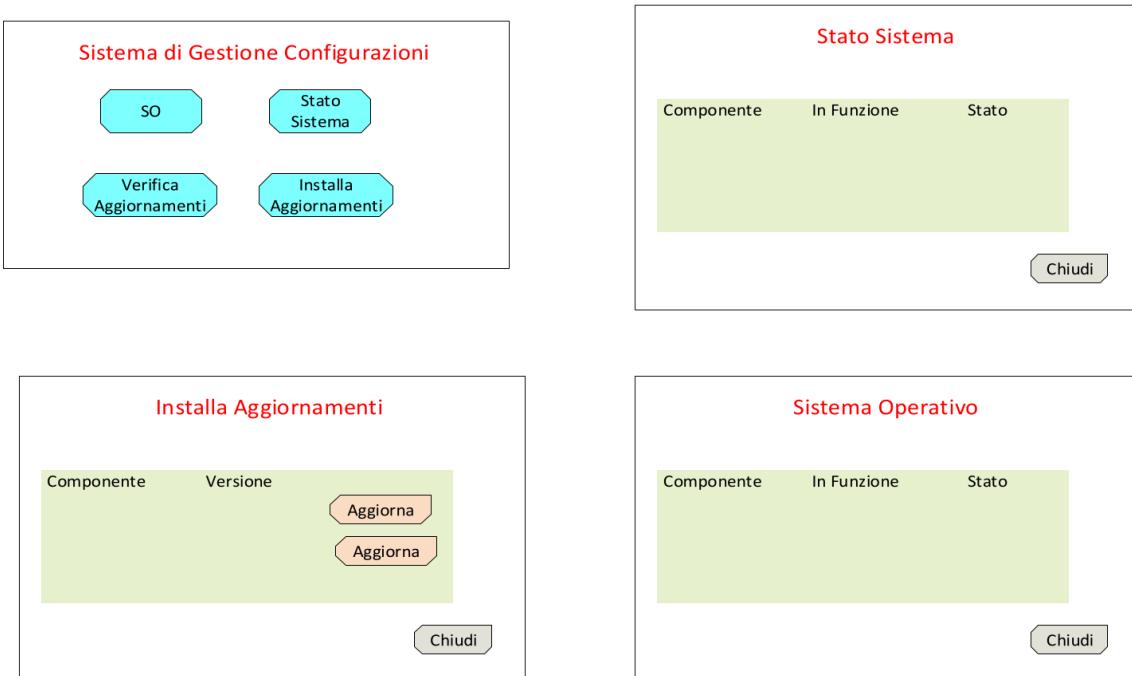
- La base di partenza di questa attività è il Piano del Collaudo sviluppato nell'analisi
- Dopo la Progettazione di Dettaglio è possibile scrivere i **test unitari di ciascuna classe**
- Successivamente vanno progettati con cura anche i **test di integrazione** del sistema

- L'output di questa attività è rappresentato dalla Suite completa dei test unitari e di integrazione

2.3.7 Progettazione per il deployment

- Seguiremo le linee guida già viste nel blocco della sicurezza:
 1. Includere supporto per visionare e analizzare le configurazioni
 2. Minimizzare i privilegi di default
 3. Localizzare le impostazioni di configurazione
 4. Fornire modi per rimediare a vulnerabilità di sicurezza

2.3.7.1 Deployment: Esempio



Lato server:

- i server dovranno essere installati su macchine all'interno di una rete privata
- la rete privata dovrà essere opportunamente protetta da un firewall a cifratura di pacchetti
- l'unico punto di contatto verso l'esterno è il Broker

2.3.8 Design Pattern

Nel 1977, Christopher Alexander disse:

«*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice»*

Parlava di costruzioni civili e di città

- La stessa frase è applicabile anche alla progettazione object-oriented
- In questo caso, le soluzioni utilizzeranno
 - oggetti, classi e interfacce
 - invece che pareti e porte...

- **Obiettivi**
 - Risolvere problemi progettuali specifici
 - Rendere i progetti object-oriented più flessibili e riutilizzabili
- Ogni design pattern
 - Cattura e formalizza l'esperienza acquisita nell'affrontare e risolvere uno specifico problema progettuale
 - Permette di riutilizzare tale esperienza in altri casi simili
- Ogni design pattern ha **quattro elementi essenziali**
 - un **nome** (significativo) - identifica il pattern
 - il **problema** - descrive quando applicare il pattern
 - la **soluzione** - descrive il pattern, cioè gli elementi che lo compongono (classi e istanze) e le loro relazioni, responsabilità e collaborazioni
 - le **conseguenze** - descrivono vantaggi e svantaggi dell'applicazione del pattern e permettono di valutare le alternative progettuali

2.3.8.1 L'importanza dei nomi dei Pattern

- Gli schemi progettuali del software hanno nomi suggestivi:
 - *Observer, Singleton, Strategy ...*
- Perché i nomi sono importanti?
 - Supportano il *chunking*, ovvero fissano il concetto nella nostra memoria e ci aiutano a capirlo
 - Facilitano la comunicazione tra progettisti

2.3.8.2 Classificazione dei Design Pattern

- **Pattern di creazione** (creational pattern)
Risolvono problemi inerenti il processo di creazione di oggetti
- **Pattern strutturali** (structural pattern)
Risolvono problemi inerenti la composizione di classi o di oggetti
- **Pattern comportamentali** (behavioral pattern)
Risolvono problemi inerenti le modalità di interazione e di distribuzione delle responsabilità tra classi o tra oggetti

Pattern di creazione	Pattern strutturali	Pattern comportamentali
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

2.3.8.3 Pattern SINGLETION

- Assicura che una classe abbia una sola istanza fornisce un punto di accesso globale a tale istanza
- La classe deve:
 - ▶ tenere traccia della sua sola istanza
 - ▶ intercettare tutte le richieste di creazione, al fine di garantire che nessuna altra istanza venga creata; sostanzialmente viene dato a chi da la richiesta di creazione il riferimento all'istanza
 - ▶ fornire un modo per accedere all'istanza unica

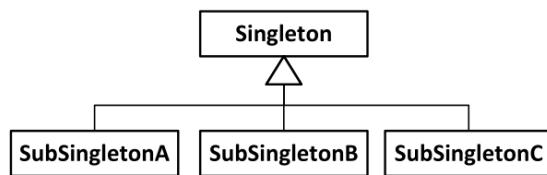
```
public class Singleton

{
    ...
    ... attributi membro di istanza ...
    private static Singleton instance = null;
    protected Singleton()
    { inizializzazione istanza }
    public static Singleton GetInstance()
    {
        if(_instance == null)
            _instance = new Singleton();
        return _instance;
    }
    ...
    ... metodi pubblici, protetti e privati ...
}
```

- Alternativa: classe non istanziabile (`static class`) con soli membri statici
 - ▶ Math
 - ▶ Convert
 - ▶ ...
- Perché un *singletone*?
- Il *singletone* **può implementare 1+ interfacce**

- Il **singleton** può essere specializzato ed è possibile creare nella GetInstance un'istanza specializzata che dipende dal contesto corrente

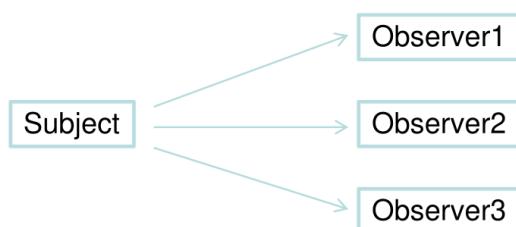
```
public static Singleton GetInstance()
{
    if(_instance == null)
        _instance = CreateInstance();
    return _instance;
}
private static Singleton CreateInstance()
{
    if(...)
        return new SubSingletonA();
    else if(...)
        return new SubSingletonB();
    else
        return new SubSingletonC();
}
```



2.3.8.4 Pattern OBSERVER

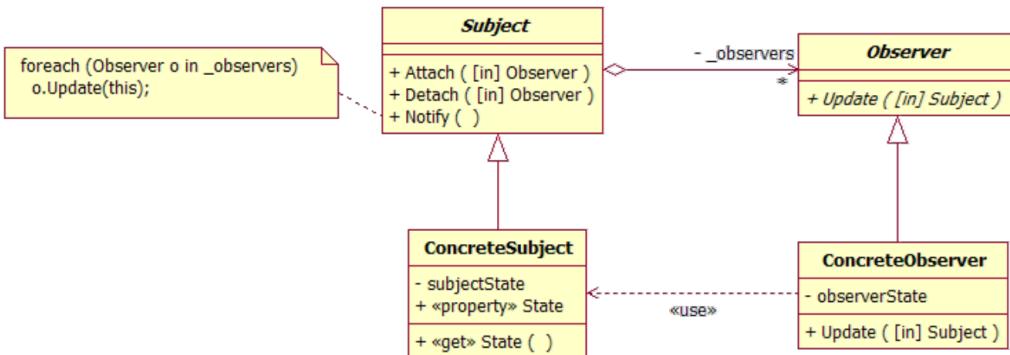
- **Contesto**

- ▶ Talvolta una modifica a un oggetto (**il soggetto**) richiede che altri oggetti (**osservatori**) siano modificati a loro volta, o quantomeno richiedano di essere notificati se il soggetto viene modificato
- ▶ Questa relazione può essere esplicitamente codificata nel soggetto, ma questo richiede che questo sappia come gli osservatori debbano essere aggiornati
 - si crea accoppiamento tra gli oggetti (**closely coupled**) e **non possono essere facilmente riusati**



- **Soluzione**

- ▶ Creare una relazione uno-a-molti più lasca tra un oggetto e gli altri che dipendono da esso
- ▶ Una modifica dell'oggetto farà sì che gli altri **ricevano una notifica**, consentendo loro di aggiornarsi di conseguenza

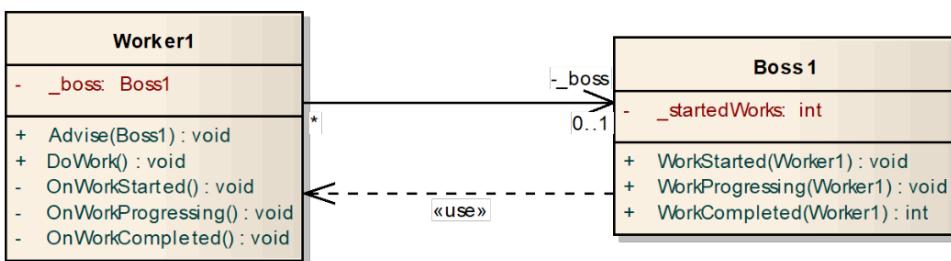


- Ogni Subject può attaccarsi o staccarsi da un Observer
- Con questo pattern il Soggetto da chi sono gli Osservatori, mentre gli Osservatori non sanno chi è il soggetto

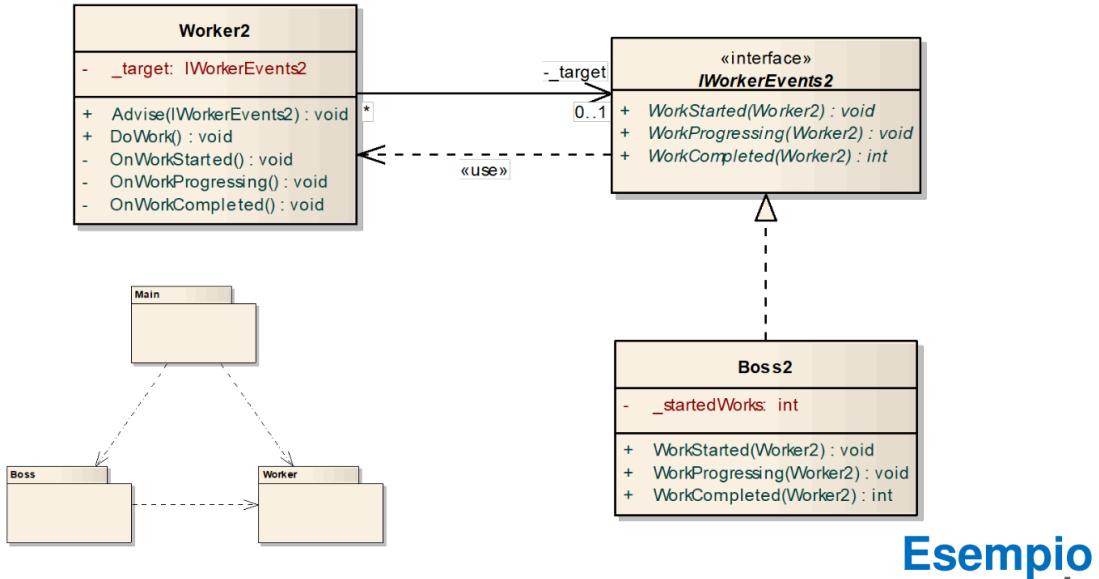
2.3.8.4.1 Esempio Boss-Worker

- È necessario modellare un'interazione tra due componenti
 - un **Worker** che effettua un'attività (o lavoro)
 - un **Boss** che controlla l'attività dei suoi Worker
- Ogni **Worker** deve notificare al proprio Boss:
 - quando il lavoro inizia
 - quando il lavoro è in esecuzione
 - quando il lavoro finisce
- Soluzioni possibili:
 1. **class-based callback relationship**
 2. **interface-based callback relationship**
 3. **pattern Observer** (lista di notifiche)
 4. **delegate-based callback relationship**
 5. **event-based callback relationship**

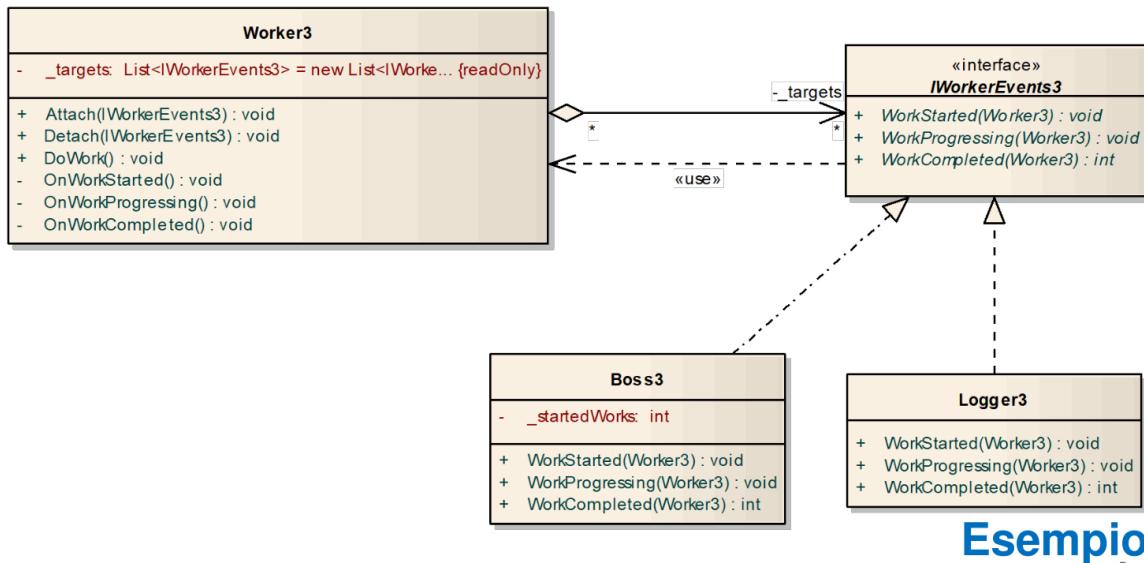
Class-based callback relationship



Interface-based callback relationship



Pattern Observer (lista di notifiche)



2.3.8.5 Pattern Model / View / Controller (MVC)

- Utilizzato per realizzare le interfacce utenti in *Smalltalk-80*
- Permette di suddividere un'applicazione, o anche la sola interfaccia dell'applicazione, in tre parti
- **Modello**: elaborazione/stato
- **View**: (o viewport) output
- **Controller**: input

Modello

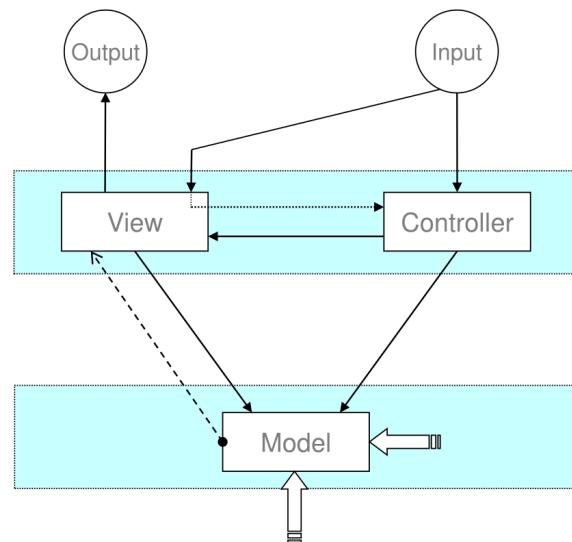
- Gestisce un insieme di dati logicamente correlati
- Risponde alle interrogazioni sui dati
- Risponde alle istruzioni di modifica dello stato
- Genera un evento quando lo stato cambia
- Registra (in forma anonima) gli oggetti interessati alla notifica dell'evento
- In Java, deve estendere la classe `java.util.Observable`

View

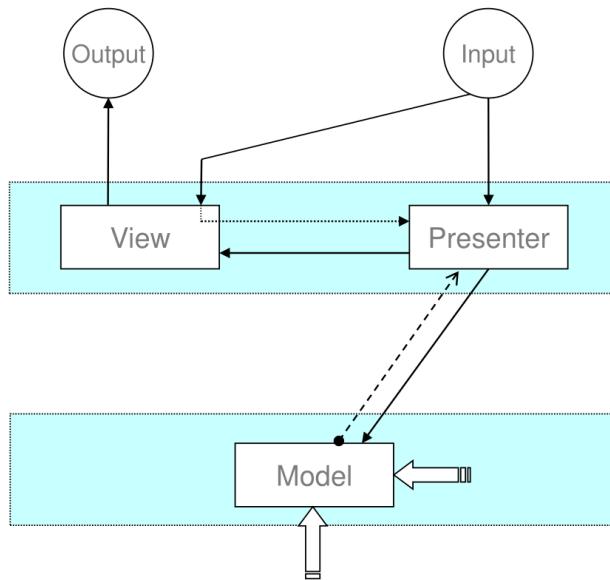
- Gestisce un'area di visualizzazione, nella quale presenta all'utente una vista dei dati gestiti dal modello
 - Mappa (parte de) i dati del modello in oggetti visuali
 - Visualizza tali oggetti su un particolare dispositivo di output
- Si registra presso il modello per ricevere l'evento di cambiamento di stato
- In Java, deve implementare l'interfaccia `java.util.Observer`

Controller

- Gestisce gli input dell'utente (mouse, tastiera, ...)
- Mappa le azioni dell'utente in comandi
- Invia tali comandi al modello e/o alla view che effettuano le operazioni appropriate
- In Java, è un *listener*



Con view passiva:



- In questo caso è il Presenter a parlare con il Model
- Posso avere diversi Presenter per lo stesso Model

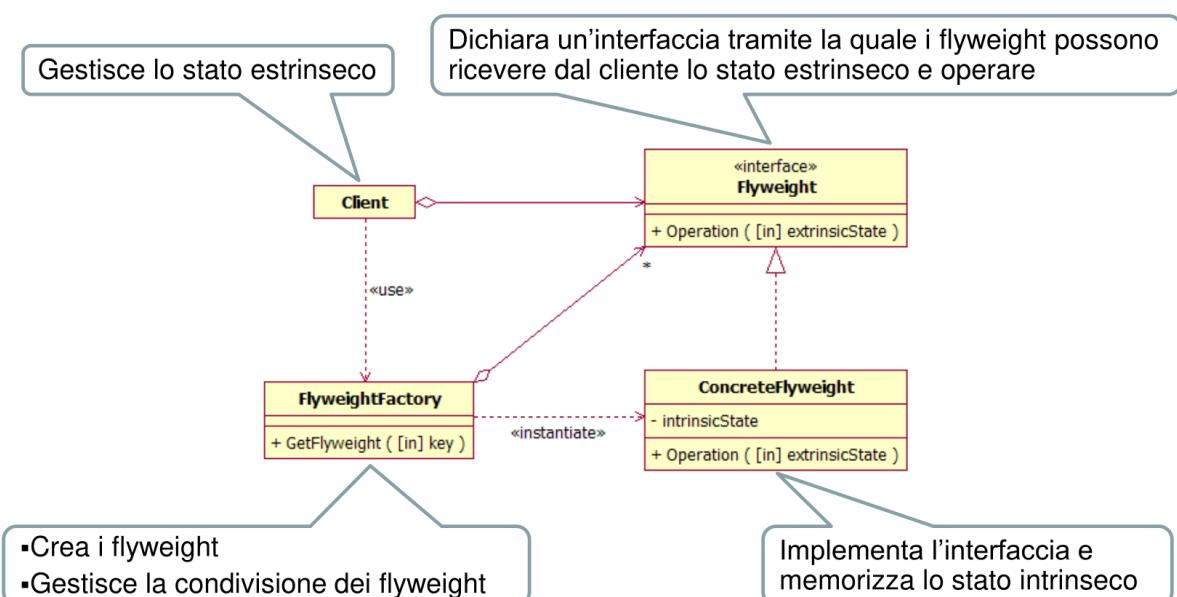
2.3.8.6 Pattern FLYWEIGHT

- Descrive come condividere oggetti "leggeri" (cioè a granularità molto fine) in modo tale che il loro uso non sia troppo costoso

- Un *flyweight* è un **oggetto condiviso** che può essere utilizzato simultaneamente ed efficientemente da più clienti (del tutto indipendenti tra loro)
- Benché condiviso, **non deve essere distinguibile da un oggetto non condiviso**
- Non deve fare ipotesi sul contesto nel quale opera
- Per assicurare una corretta condivisione, i clienti
 - non devono istanziare direttamente i flyweight
 - ma devono ottenerli esclusivamente tramite una **FlyweightFactory**

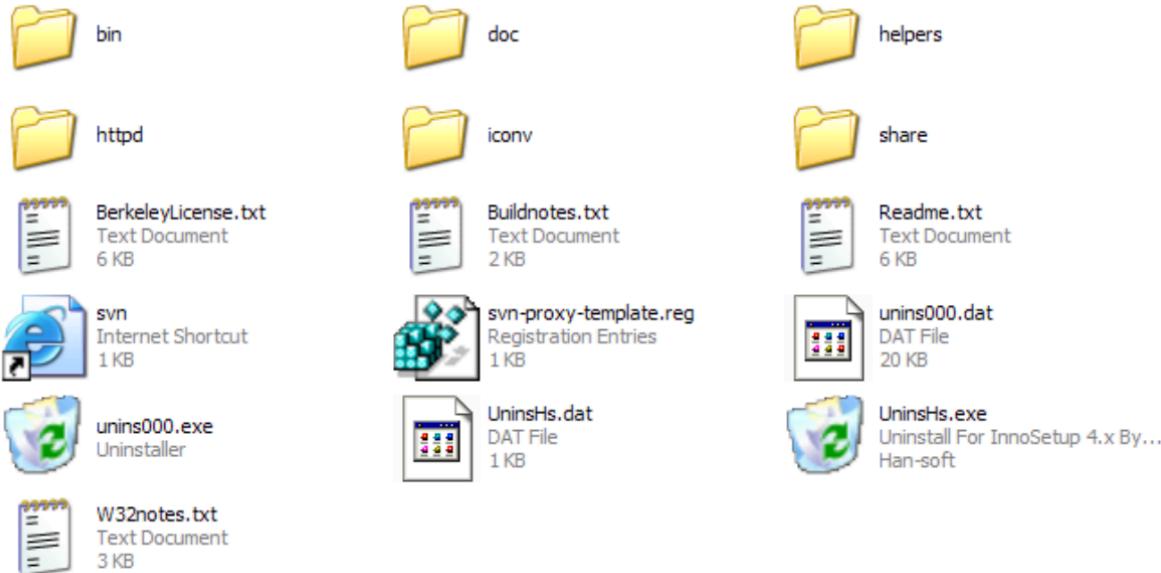
```
private Dictionary<DictionaryKeyType, FlyweightType> flyweights;
...
public FlyweightType GetFlyweight(KeyType key)
{
    if(!flyweights.ContainsKey(key))
    {
        flyweights.Add(key, CreateFlyweight(key));
    }
    return flyweights[key];
}
```

- Distinzione tra stato intrinseco e stato estrinseco
- **Stato intrinseco:**
 - **Non dipende dal contesto di utilizzo** e quindi **può essere condiviso** da tutti i clienti
 - Memorizzato nel *flyweight*
- **Stato estrinseco**
 - **Dipende dal contesto di utilizzo** e quindi **non può essere condiviso** dai clienti
 - Memorizzato nel cliente o calcolato dal cliente
 - Viene passato al *flyweight* quando viene invocata una sua operazione



2.3.8.6.1 Esempio

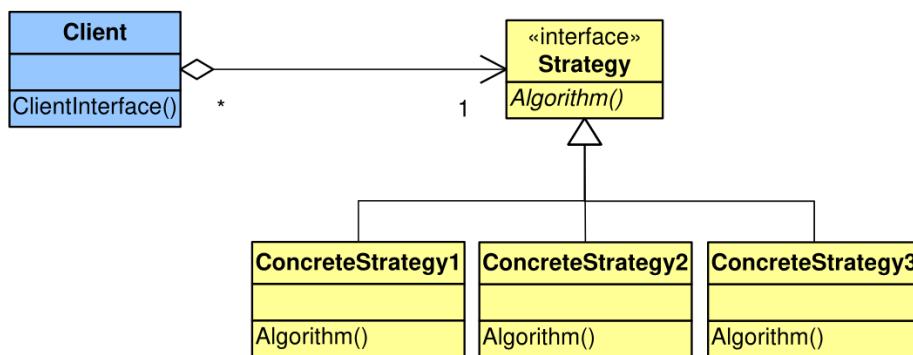
- Si supponga di usare il pattern flyweight per condividere delle icone tra vari clienti

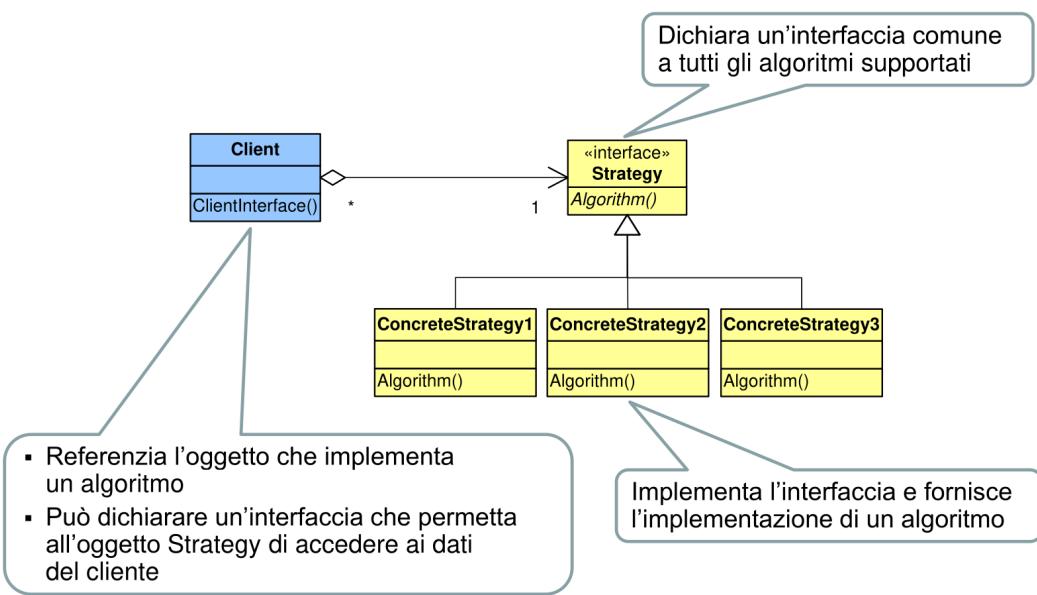


- Lo **stato intrinseco** (memorizzato nel flyweight) comprenderà tutte le informazioni che i clienti devono (e possono) condividere:
 - Nome dell'icona
 - Bitmap dell'icona
 - Dimensioni originali, ...
- Lo **stato estrinseco** (memorizzato nel cliente) comprenderà il contesto in cui l'icona dovrà essere disegnata (dipendente dal singolo cliente):
 - Posizione dell'icona
 - Dimensioni richieste, ...

2.3.8.7 Pattern STRATEGY

- Permette di
 - definire un insieme di algoritmi tra loro correlati,
 - incapsulare tali algoritmi in una gerarchia di classi e
 - rendere gli algoritmi intercambiabili

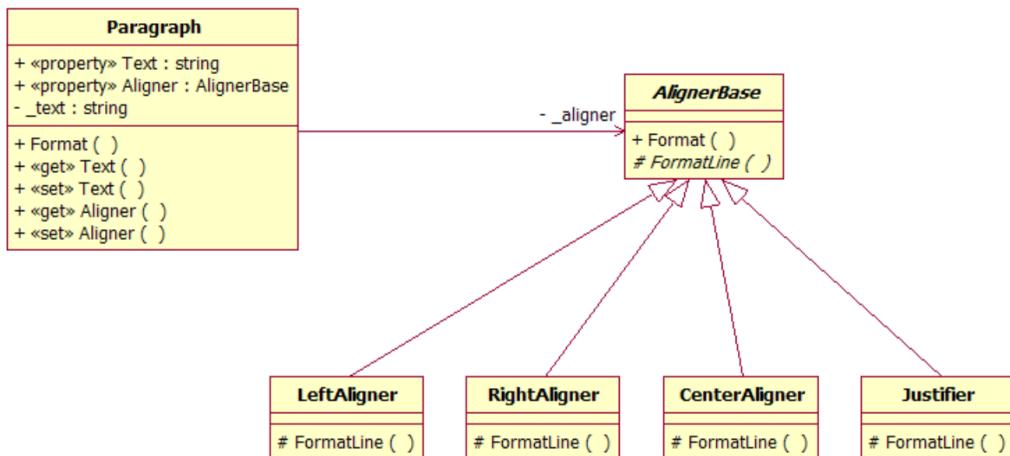




2.3.8.7.1 Esempio

- **Allineamento del testo di un paragrafo**

Esistono politiche diverse di allineamento



- **AlignerBase**

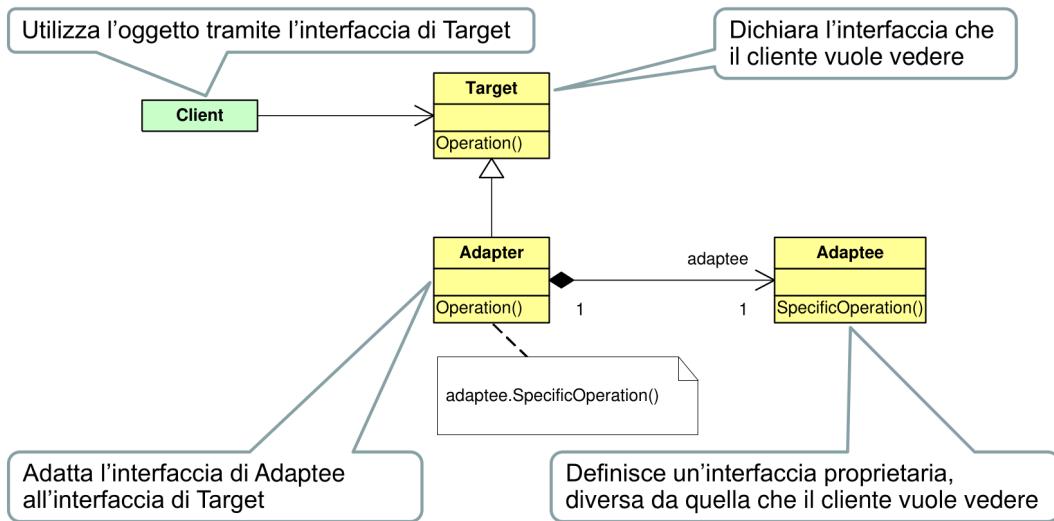
- suddivide il testo in linee (**Format**)
- delega alle sue sottoclassi l'allineamento delle singole linee (**FormatLine**)
- Paragraph utilizza i servizi di un “*Aligner*” specificato dinamicamente run-time
- È possibile realizzare gli “*Aligner*” utilizzando il pattern ***flyweight***

Esempio

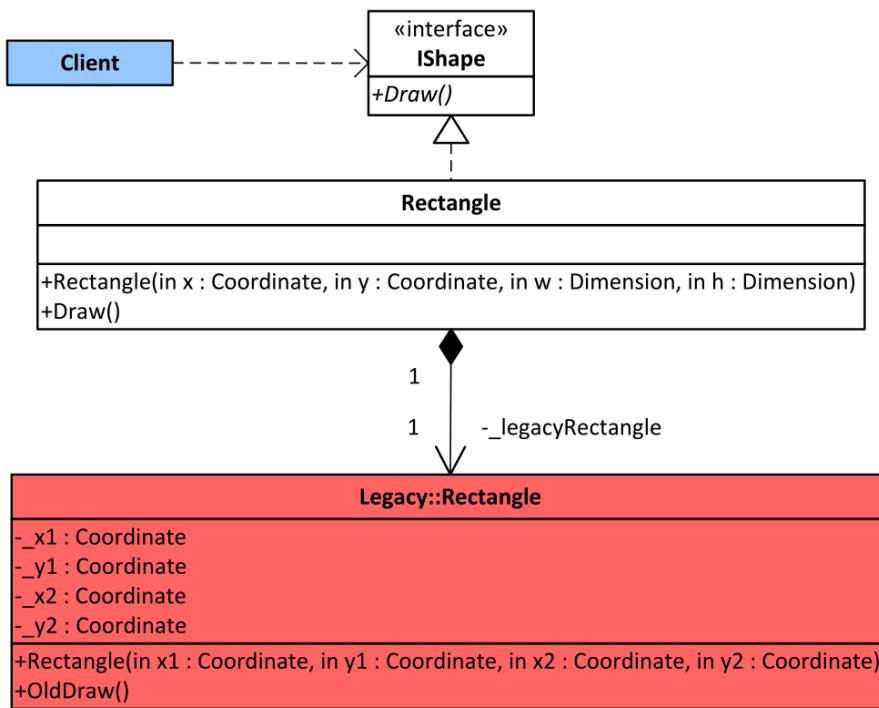
2.3.8.8 Pattern ADAPTER

- Converte l’interfaccia originale di una classe nell’interfaccia (diversa) che si aspetta il cliente
- Permette a classi che hanno interfacce incompatibili di lavorare insieme
- Si inserisce, tra le due classi, l’interfaccia Adapter, che ingloba la classe che deve erogare i servizi, in modo che possa comunicare con la classe che li deve usare
- Si usa quando

- ▶ si vuole riutilizzare una classe esistente
- ▶ la sua interfaccia non è conforme a quella desiderata
- Noto anche come **wrapper**
- Si realizza con composizione-delega



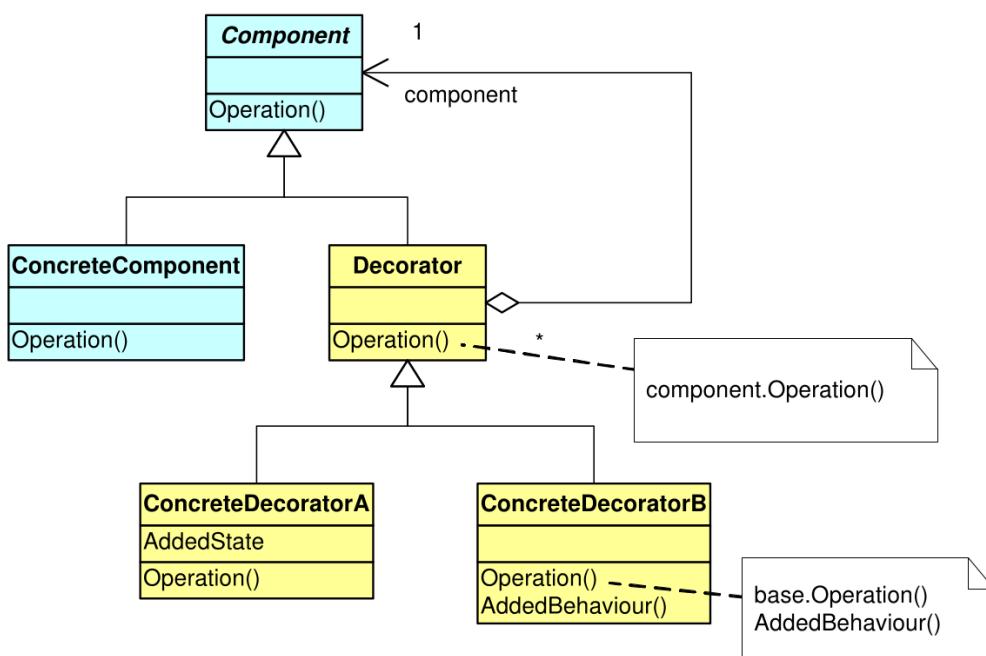
2.3.8.8.1 Esempio



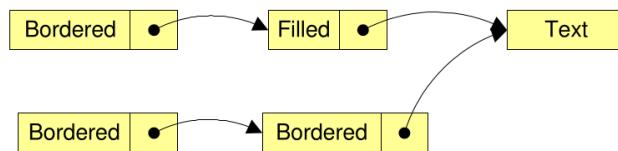
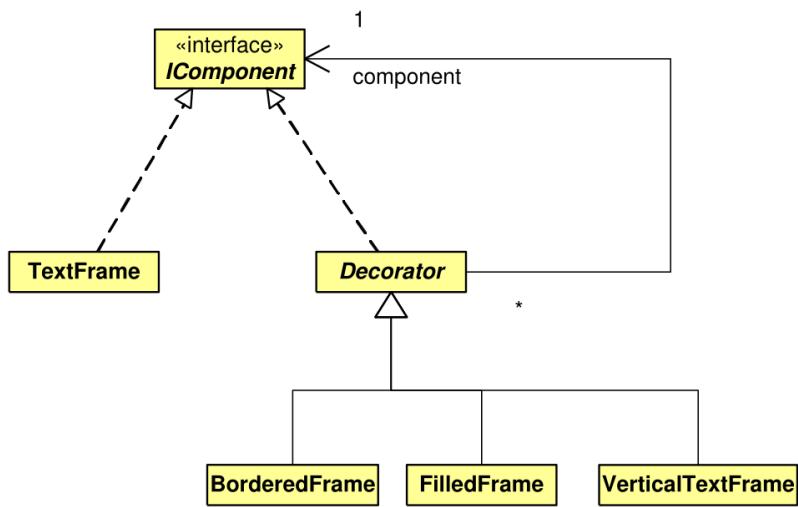
2.3.8.9 Pattern DECORATOR

- Permette di **aggiungere responsabilità** a un oggetto dinamicamente
- Fornisce un'**alternativa flessibile alla specializzazione**
 - ▶ In alcuni casi, le estensioni possibili sono talmente tante che per poter supportare ogni possibile combinazione, si dovrebbe definire un numero troppo elevato di sottoclassi
- TextBox
 - ▶ BorderTextBox

- ▶ FilledTextBox
- ▶ VerticalTextBox
- ▶ BorderFilledTextBox
- ▶ BorderVerticalTextBox
- ▶ BorderFilledVerticalTextBox
- ▶ FilledVerticalTextBox
- E se volessi
 - ▶ 2 o più bordi
 - ▶ Cambiare il font
 - ▶ ...

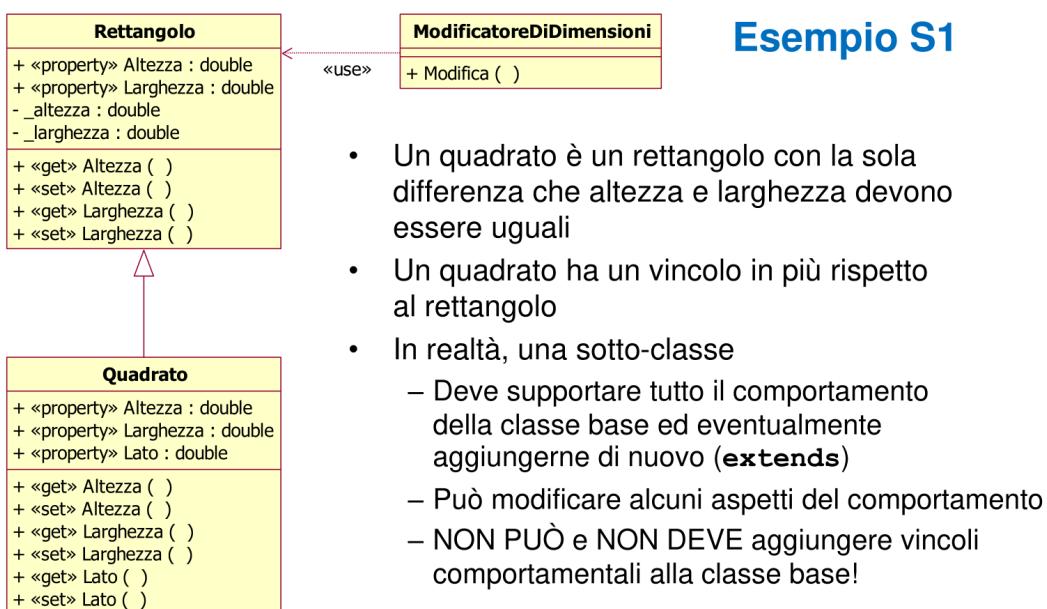


- **Component (interfaccia o classe astratta)**
 - ▶ Dichiara l’interfaccia di tutti gli oggetti ai quali deve essere possibile aggiungere dinamicamente responsabilità
- **ConcreteComponent**
 - ▶ Definisce un tipo di oggetto al quale deve essere possibile aggiungere dinamicamente responsabilità
- **Decorator (classe astratta)**
 - ▶ Mantiene un riferimento a un oggetto di tipo Component e definisce un’interfaccia conforme all’interfaccia di Component
- **ConcreteDecorator**
 - ▶ Aggiunge responsabilità al componente referenziato

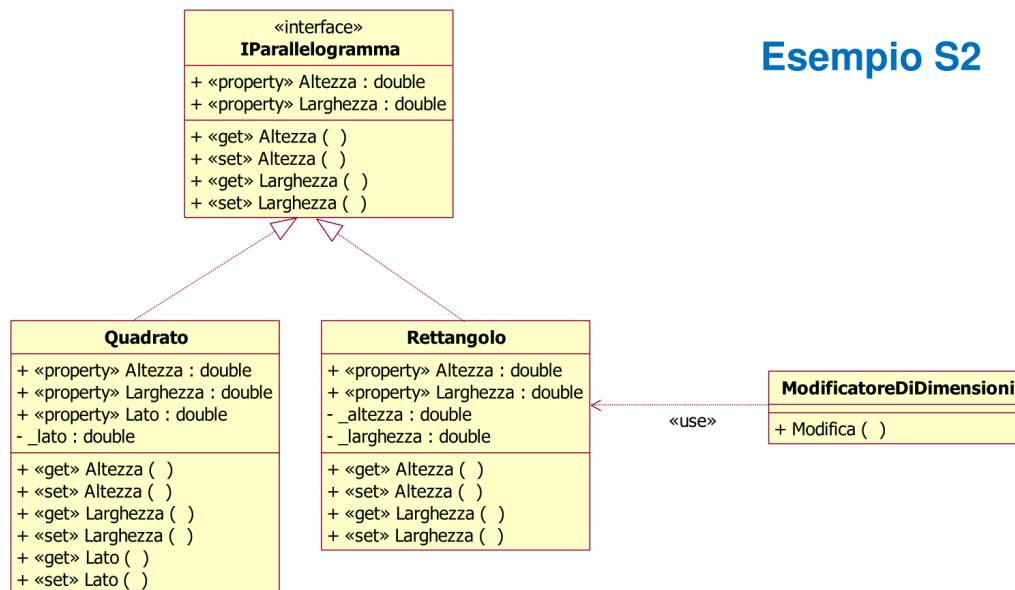


2.3.8.10 Ereditarietà Dinamica

- Una sotto-classe deve sempre essere una **versione più specializzata** della sua super-classe (o classe base)
- Un buon test sul corretto utilizzo dell'ereditarietà è che sia valido il **principio di sostituibilità di Liskov**:
“B è una sotto-classe di A se e solo se ogni programma che utilizzi oggetti di classe A può utilizzare oggetti di classe B senza che il comportamento logico del programma cambi”
- Perché ciò sia valido, è necessario che:
 - le **pre-condizioni** di tutti i metodi della sotto-classe siano uguali o più deboli
 - le **post-condizioni** di tutti i metodi della sotto-classe siano uguali o più forti
 - ogni metodo ridefinito nella sotto-classe deve mantenere la **semantica** del metodo originale



- Il metodo `Modifica` della classe `ModificatoreDiDimensioni`
 - ▶ funziona correttamente su un `Rettangolo`
 - ▶ ma NON funziona correttamente su un `Quadrato`
- Quindi non è possibile passare un'istanza di `Quadrato` dove è prevista un'istanza di `Rettangolo`
 - ▶ il principio di sostituibilità di Liskov è violato!
- **Conclusione:** un quadrato NON è un rettangolo perché pone dei nuovi vincoli al concetto di rettangolo
- Come possiamo tenere conto di ciò che il rettangolo e il quadrato hanno in comune?



- Cosa intendiamo esattamente per `Rettangolo` e per `Quadrato`?
- **Rettangolo:** parallelogramma i cui angoli sono retti
- **Parallelogramma:** quadrilatero i cui lati opposti sono paralleli tra loro
- **Quadrilatero:** poligono avente quattro lati e quattro angoli
 - ▶ Quadrilateri notevoli sono il quadrato, il rettangolo, il parallelogramma, il rombo e il trapezio
- **Polygono:** figura geometrica limitata da una linea poligonale chiusa
- **Rombo:** parallelogramma equilatero in cui gli angoli adiacenti sono diversi tra loro
- **Quadrato:** parallelogramma equilatero ed equiangolo
- Cosa intendiamo esattamente per `Rettangolo` e per `Quadrato` nella nostra applicazione?
- **Ipotesi:** abbiamo a che fare esclusivamente con parallelogrammi

1. Lati e angoli NON sono modificabili

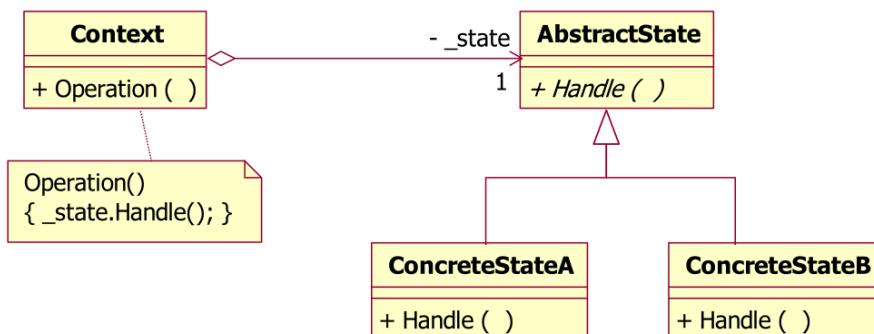
- Definire quattro classi concrete che derivano dalla classe astratta `Parallelogramma` (o implementano `IParallelogramma`): `Rettangolo` , `Quadrato` , `Rombo` , `ParallelogrammaGenerico`

- Usare una factory che in base ai valori dei lati e degli angoli istanzia un rettangolo (che NON deve avere i lati uguali), un quadrato, un rombo o un parallelogramma generico

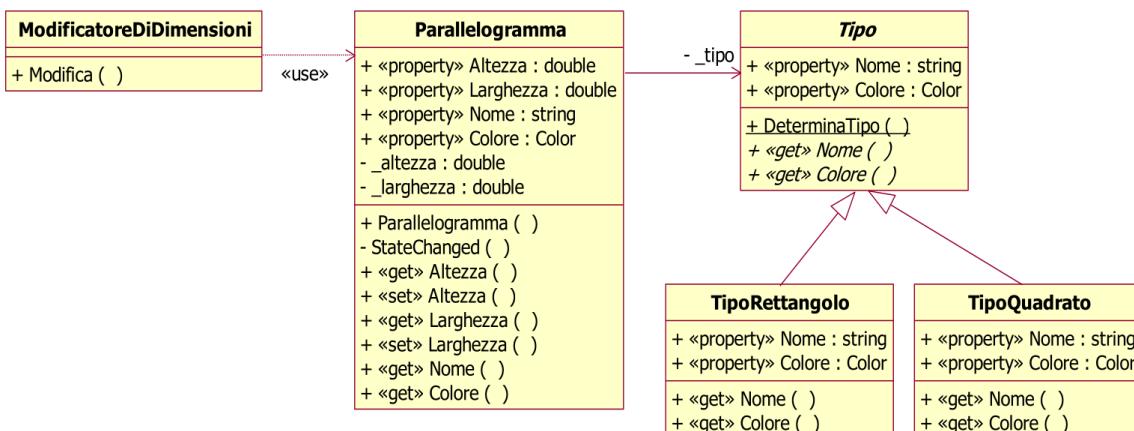
2. Lati e angoli sono modificabili

- Definire un'unica classe concreta **Parallelogramma** le cui istanze possono comportarsi a seconda del loro stato come: un rettangolo, un quadrato, un rombo, o un parallelogramma generico
- Come può un oggetto cambiare comportamento, al cambiare del suo stato?
- 1 possibilità: si cambia la classe dell'oggetto run-time
 - ▶ nella maggior parte dei linguaggi di programmazione a oggetti, questo non è possibile (inoltre, è meglio che un oggetto non possa cambiare classe durante la sua esistenza)
- la classe di un oggetto deve basarsi sulla sua essenza e non sul suo stato
- 2 possibilità: si utilizza il pattern State che usa un **meccanismo di delega**, grazie al quale l'oggetto è in grado di comportarsi **come se** avesse cambiato classe

2.3.8.11 Pattern STATE

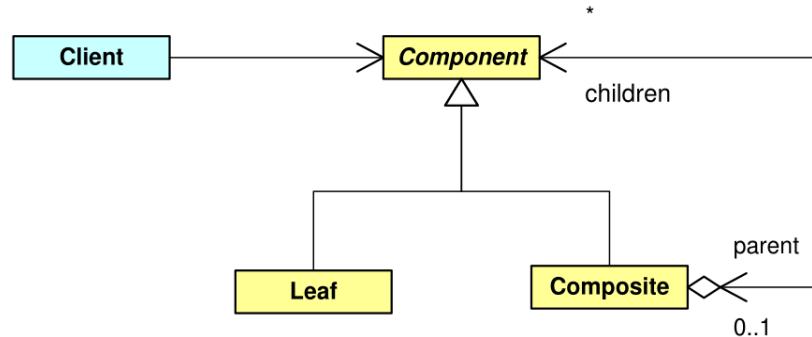


- Localizza il comportamento specifico di uno stato e suddivide il comportamento in funzione dello stato
- Le classi concrete contengono la logica di transizione da uno stato all'altro
- Permette anche di emulare l'ereditarietà multipla

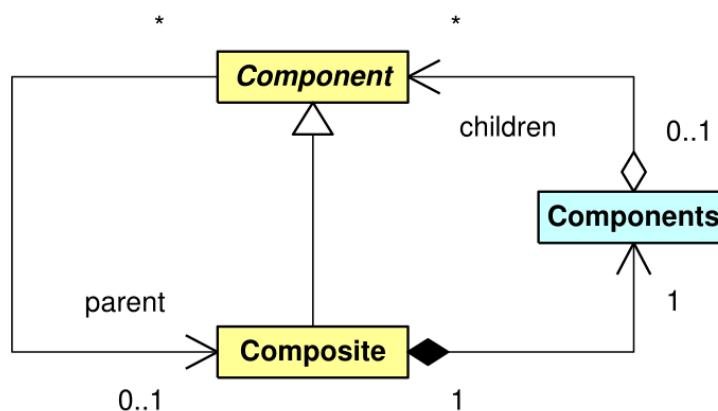


2.3.8.12 Pattern COMPOSITE

- Permette di comporre oggetti in una **struttura ad albero**, al fine di rappresentare una **gerarchia di oggetti contenitori-oggetti contenuti**
- Permette ai clienti di **trattare in modo uniforme oggetti singoli e oggetti composti**



- **Component** (classe astratta)
 - Dichiara l'interfaccia
 - Realizza il comportamento di default
- **Client**
 - Accede e manipola gli oggetti della composizione attraverso l'interfaccia di **Component**
- **Leaf**
 - Descrive oggetti che non possono avere figli -foglie
 - Definisce il comportamento di tali oggetti
- **Composite**
 - Descrive oggetti che possono avere figli -contenitori
 - Definisce il comportamento di tali oggetti



- Il contenitore dei figli deve essere un attributo di **Composite** e può essere di qualsiasi tipo (array, lista, albero, tabella hash, ...)

- **Riferimento esplicito al genitore** (parent)
 - Semplifica l'attraversamento e la gestione della struttura

- ▶ L'attributo che contiene il riferimento al genitore e la relativa gestione devono essere posti nella classe **Component**
 - **Invariante**
 - ▶ Tutti gli elementi che hanno come *parent* lo stesso componente devono essere (gli unici) figli di quel componente
 - encapsulare l'assegnamento di parent nei metodi Add e Remove della classe Composite , oppure
 - encapsulare le operazioni di Add e Remove nella set dell'attributo *parent* della classe Component
- ```

public class Composite : Component
{
 ...
 public void Add(Component aChild)
 {
 if(aChild.Parent != null)
 throw new ArgumentException(...);

 _children.Add(aChild);
 aChild._parent = this;
 }
 ...
}

public class Composite : Component
{
 ...
 public void Remove(Component aChild)
 {
 if(aChild.Parent != this)
 throw new ArgumentException(...);

 if(!_children.Contains(aChild))
 throw new ArgumentException(...);

 _children.Remove(aChild);
 }
}

```

```

 aChild._parent = null;
}

...
}

public class Component
{

 ...
 public Composite Parent
 {
 get { return _parent; }
 set
 {
 if(value != _parent)
 {
 if(_parent != null)
 _parent.Remove(this);
 if(value != null)
 value.Add(this);
 }
 }
 }
 ...
}

```

- **Massimizzazione dell'interfaccia Component**

- ▶ Un obiettivo del pattern Composite è quello di fare in modo che il cliente veda solo l'interfaccia di Component: in Component devono essere inserite tutte le operazioni che devono essere utilizzate dai clienti
  - nella maggior parte dei casi, Component definisce una realizzazione di default che le sotto classi devono ridefinire
- Alcune di queste operazioni possono essere prive di significato per gli oggetti foglia (Add ,Remove , ...)
- **Trasparenza**
  - ▶ Dichiaro tutto al livello più alto, in modo che il cliente possa trattare gli oggetti in modo uniforme ma... **il cliente potrebbe cercare di fare cose senza senso**, come aggiungere figli alle foglie
- Se scegliamo la trasparenza
  - ▶ Add e Remove devono avere una realizzazione di default che genera un'eccezione
  - ▶ dovremmo disporre di un modo per verificare se è possibile aggiungere figli all'oggetto su cui si vuole agire

// Il cliente conosce solo Component

```

Component parent = ComponentFactory.CreateInstance(...);
...
Component child = ComponentFactory.CreateInstance(...);
...
// Prima di inserire un figlio,
// occorre controllare se è possibile

if(parent.IsComposite())
 parent.Add(child);

```

- **Sicurezza**

- Tutte le operazioni sui figli vengono messe in Composite - a questo punto, qualsiasi invocazione sulle foglie genera un errore in fase di compilazione ma... **il cliente deve conoscere e gestire due interfacce differenti**
- Se scegliamo la sicurezza
  - dobbiamo disporre di un modo per verificare se l'oggetto su cui si vuole agire è un Composite

```
// Il cliente conosce Component e Composite
```

```

Component child = ComponentFactory.CreateComponent(...);
Composite parent1 = ComponentFactory.CreateComposite(...);
parent1.Add(child);
...
Component parent2 = ComponentFactory.CreateComponent(...);
// Errore di compilazione
parent2.Add(child);
// Prima di inserire un figlio,
// occorre controllare se è possibile e fare un cast

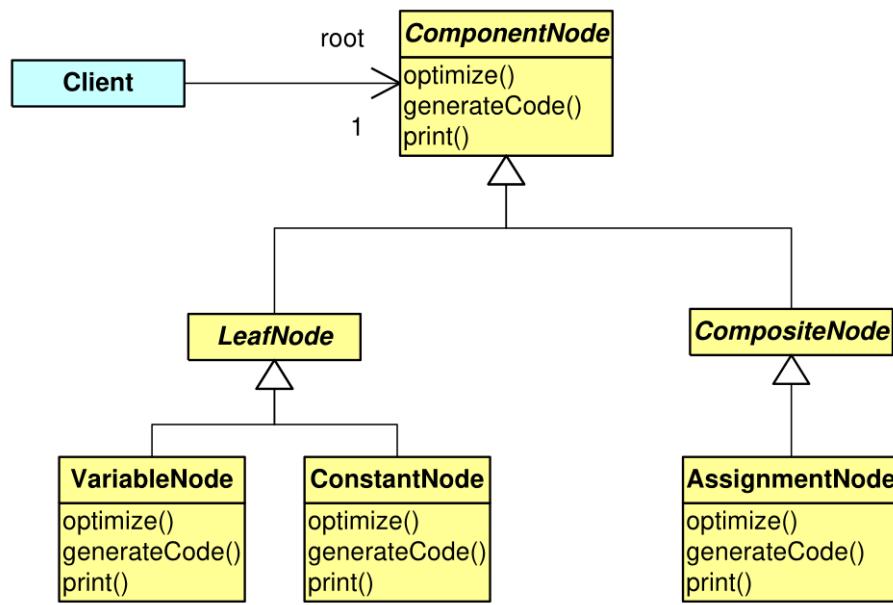
if(parent2 is Composite)
 ((Composite) parent2).Add(child);

```

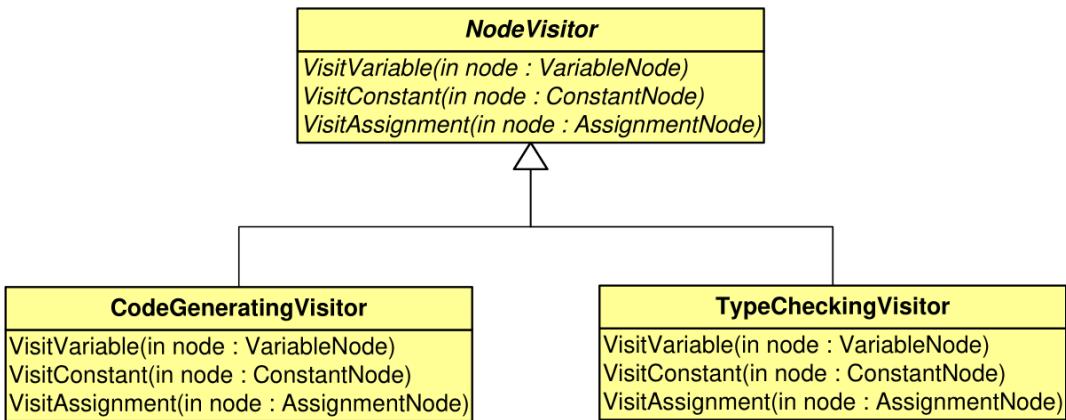
### 2.3.8.13 Pattern VISITOR

- Permette di **definire una nuova operazione** da effettuare su gli elementi di una struttura, **senza dover modificare le classi degli elementi coinvolti**
- Ad esempio, si consideri la rappresentazione di un programma come “**abstract syntax tree**” (AST) - i cui nodi descrivono elementi sintattici del programma
- Su tale albero devono poter essere effettuate molte operazioni di tipo diverso
  - Controllare che tutte le variabili siano definite
  - Eseguire delle ottimizzazioni
  - Generare il codice macchina
  - Stampare l'albero in un formato leggibile
  - ...

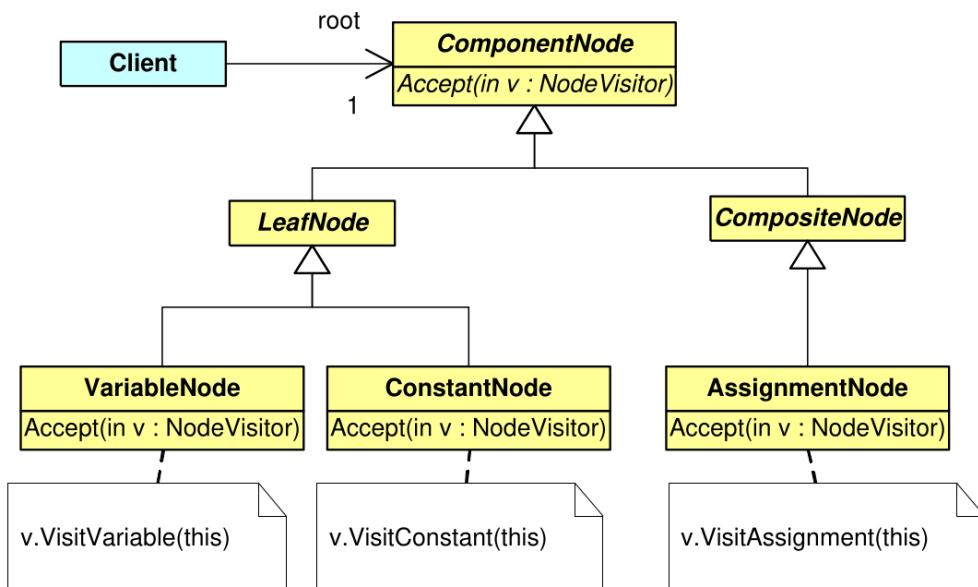
Per l'AST utilizziamo il *pattern Composite*



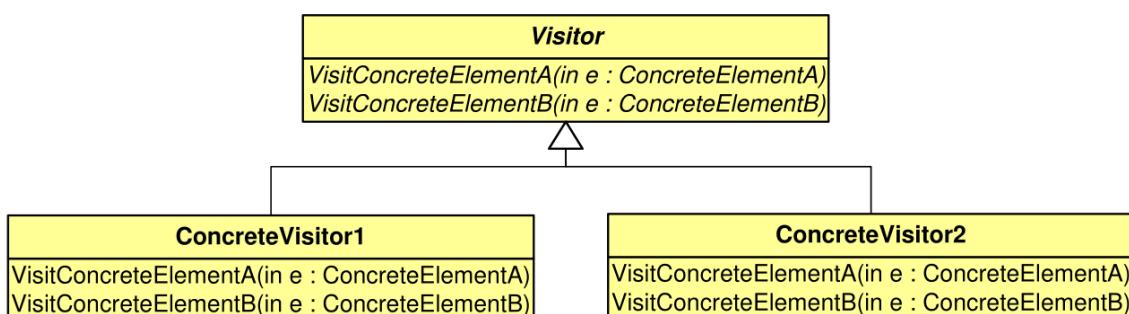
- In seguito potremmo voler effettuare **altri tipi di operazioni**
  - controllare che le variabili siano state inizializzate prima dell'uso
  - ristrutturare automaticamente il programma
  - calcolare varie metriche
  - ...
- Se distribuiamo le operazioni sui vari tipi di nodo, otteniamo un sistema che è difficile da
  - capire
  - modificare
  - estendere
- La soluzione è quella di eliminare le singole operazioni dall'AST (la cui responsabilità principale è quella di rappresentare un programma sotto forma di albero)
- **Tutto il codice relativo ad un singolo tipo di operazione** (ad es., generazione del codice) viene raccolto in **una singola classe**
- I nodi dell'AST devono **accettare la visita** delle istanze di queste nuove classi (**visitor**)
- Per aggiungere un **nuovo tipo di operazione**, è sufficiente progettare una **nuova classe**
- Il Visitor deve dichiarare **un'operazione per ogni tipo di nodo** concreto



- Ogni nodo deve dichiarare **un'operazione per accettare un generico visitor**

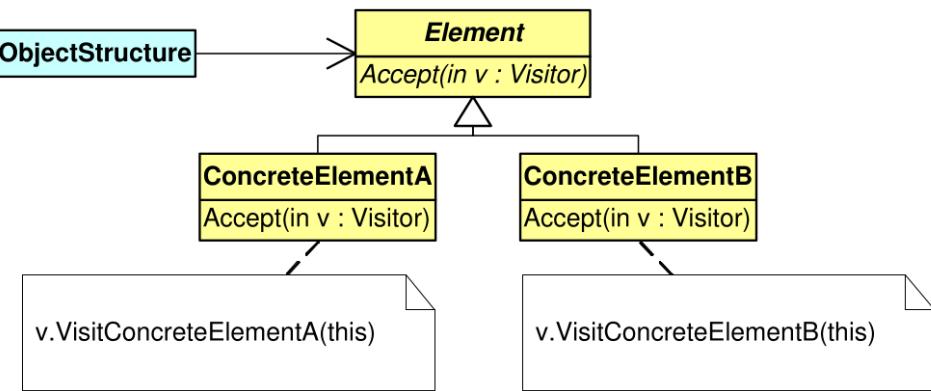


- Visitor** (classe astratta o interfaccia)
  - Dichiara un metodo **Visit** per ogni classe di elementi concreti
- ConcreteVisitor**
  - Definisce tutti i metodi **Visit**
  - Globalmente **definisce l'operazione da effettuare sulla struttura** e (se necessario) ha un proprio stato



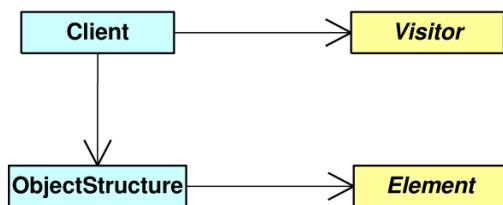
- Element** (classe astratta o interfaccia)
  - Dichiara un metodo **Accept** che accetta un Visitor come argomento
- ConcreteElement**

- Definisce il metodo [Accept](#)



- **ObjectStructure**

- Può essere realizzata come Composite o come normale collezione (array, lista, ...)
- Deve poter enumerare i suoi elementi
- Deve dichiarare un'interfaccia che permetta a un cliente di far visitare la struttura a un Visitor



- **Facilita l'aggiunta di nuove operazioni**

- È possibile aggiungere nuove operazioni su una struttura esistente, semplicemente aggiungendo un nuovo visitor concreto
- Senza il pattern Visitor, sarebbe necessario aggiungere un metodo ad ogni classe degli elementi della struttura

- Ogni Visitor concreto

- Raggruppa i metodi necessari a eseguire una data operazione
- Nasconde i dettagli di come tale operazione debba essere eseguita

- **Incapsulamento**

- Ogni Visitor deve essere in grado di accedere allo stato degli elementi su cui deve operare

- È difficile aggiungere una nuova classe **ConcreteElement**

- Per ogni nuova classe **ConcreteElement** è necessario inserire un nuovo metodo `Visit` in tutti i **Visitor** esistenti
  - la gerarchia **Element** deve essere poco o per nulla modificabile - cioè essere **stabile**

- **Visita di elementi non correlati**

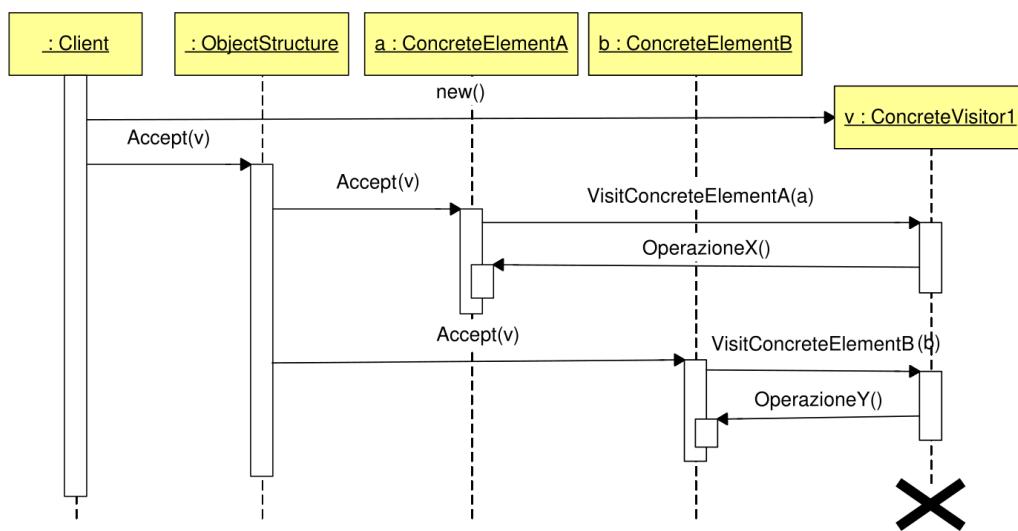
- Non è necessario che tutti gli elementi da visitare derivino da una classe comune
- ```

VisitClasseA(ClasseGerarchiaA a);
VisitClasseB(ClasseGerarchiaB b);
  
```

- **Stato**

- Durante l'operazione ogni Visitor può modificare il proprio stato - ad esempio, per accumulare dei valori o altro

```
public class CompositeElement : Element
{
    ...
    private List<Element> _children;
    ...
    public override void Accept(Visitor visitor)
    {
        foreach (Element aChild in _children)
            aChild.Accept(visitor);
        visitor.VisitCompositeElement(this);
    }
    ...
}
```



- **Double dispatch**

- L'operazione che deve essere effettuata dipende dal tipo di due oggetti
- il visitor
- l'elemento

- **Accept** è un'operazione di tipo **double dispatch**

Esempio + EsempioDecorator

2.3.8.14 Anti Pattern

- Oltre ai pattern utili esistono anche gli anti-pattern, che descrivono situazioni ricorrenti e soluzioni notoriamente dannose
- Esempio: *Interface Bloat*, che consiste nell'aggiungere così tante funzionalità a un'interfaccia da renderla impossibile da implementare (o usare!)
- Sostanzialmente, sono soluzioni che non soddisfano i design principle!

2.3.8.15 Pattern ABSTRACT FACTORY

- **Problema:**

- creazione di oggetti connessi o dipendenti tra loro, senza bisogno che il client debba specificare i nomi delle classi concrete all'interno del proprio codice
- esempio:

```

Menu m;
if (style == Macos) m = new MacosMenu;
else if(style == ...) m = new...

```

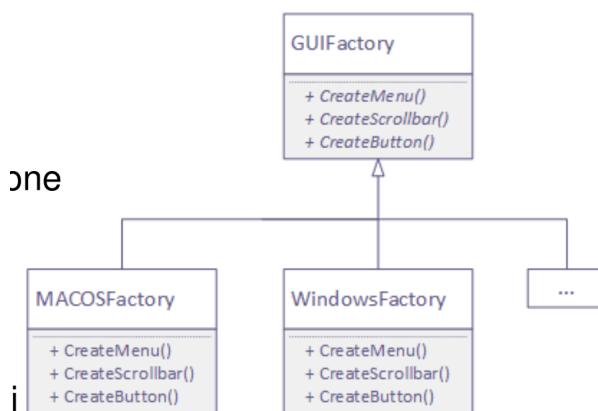
- la stessa cosa va ripetuta per pulsanti, view...

- **Requisito:**

- si vuole un sistema indipendente da come gli oggetti vengono creati, composti e rappresentati
- si vuole permettere la configurazione del sistema come scelta tra diverse famiglie di prodotti
- si vuole che i prodotti che sono organizzati in famiglie siano vincolati ad essere utilizzati con prodotti della stessa famiglia

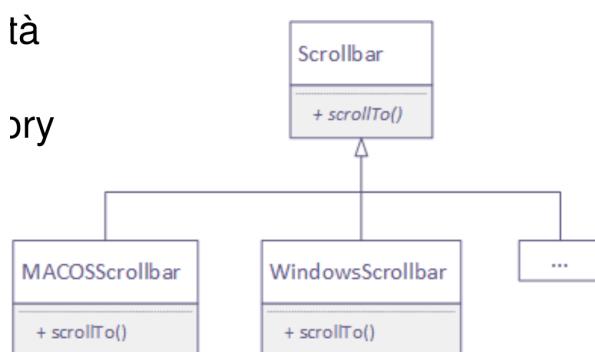
- **Soluzione:**

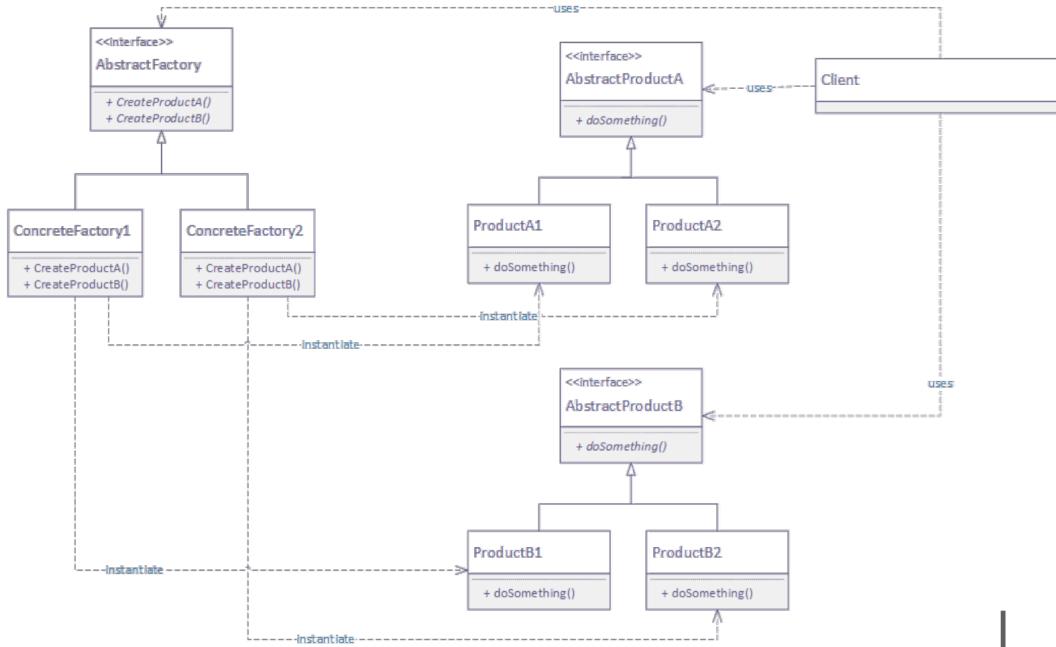
- Definizione di una classe
 - astrae la creazione di una famiglia di oggetti
 - istanze diverse costituiscono implementazioni diverse di membri di tale famiglia



- **Soluzione:**

- La creazione dei prodotti è responsabilità delle classi **ConcreteFactory**





- **Conseguenze:**

- isola le classi concrete
 - il codice successivo all'istanziazione è indipendente dalla classe concreta
- consente di cambiare in modo semplice la famiglia di prodotti utilizzata
 - la coerenza col resto del codice è assicurata dall'utilizzo delle interfacce astratte e non delle classi concrete, secondo l'OCP
- promuove la coerenza nell'utilizzo dei prodotti

- **Conseguenze:**

- difficile aggiungere supporto per nuove tipologie di prodotti
 - Dato che `AbstractFactory` definisce tutte le varie tipologie di prodotti che è possibile istanziare, aggiungere una tipologia richiede di modificare l'interfaccia della factory

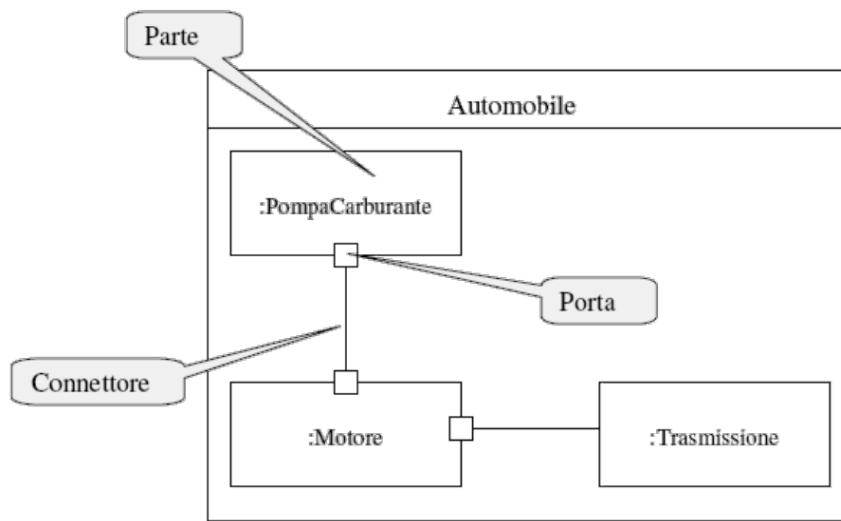
2.4 Diagramma dei componenti e di Deployment

2.4.1 Diagramma dei Componenti

- Da UML 2.0 il concetto di componente si è evoluto rispetto alla versione precedente dello standard
- Specifica un **contratto formale di servizi offerti e richiesti** in termini di interfacce (eventualmente esposte tramite porte)
- Il concetto di componente è strettamente legato a quello di **struttura composita** che spesso viene impiegata per rappresentare le parti interne del componente
- Un componente è tipicamente specificato da uno o più classificatori (ad es. classi) e può essere implementato da uno o più artefatti (file eseguibile, script, ...)
- Gli **internals** (parti interne) sono inaccessibili solo attraverso le interfacce

2.4.1.1 Struttura Composita

- Il Diagramma di Struttura Composita ha l'obiettivo di rappresentare la struttura interna (le parti) di un classificatore (classe, componente...), inclusi i punti di interazione (porte) utilizzati per accedere alle caratteristiche della struttura

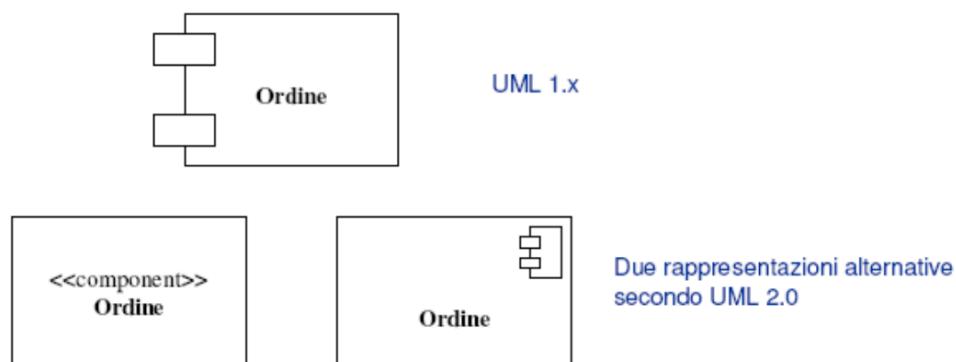


- Introdotto per scomporre gerarchicamente un classificatore, mostrandone la struttura interna:
 - mostra la struttura interna di un classificatore complesso
 - mostra in modo separato l'**interfaccia** di un classificatore dalla sua **struttura interna**
 - descrive i ruoli che i diversi elementi della struttura giocano per soddisfare l'obiettivo della struttura stessa e le interazioni richieste
- Questo permette al progettista di prendere un oggetto complesso e spezzarlo in parti più piccole e semplici
- È una sorta di strumento di zoom utile per gestire la complessità di rappresentazione

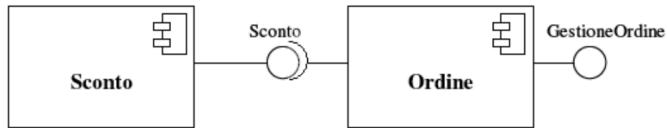
2.4.1.2 Package vs. Struttura Composita

- Per capire bene la differenza tra i package e le strutture composite bisogna pensare che
 - i primi rappresentano un raggruppamento logico al momento dell'analisi
 - mentre le seconde fanno riferimento a quello che succede durante l'esecuzione
- Di conseguenza le strutture composite sono adatte a rappresentare i componenti e le loro parti, e sono usate spesso nei diagrammi dei componenti

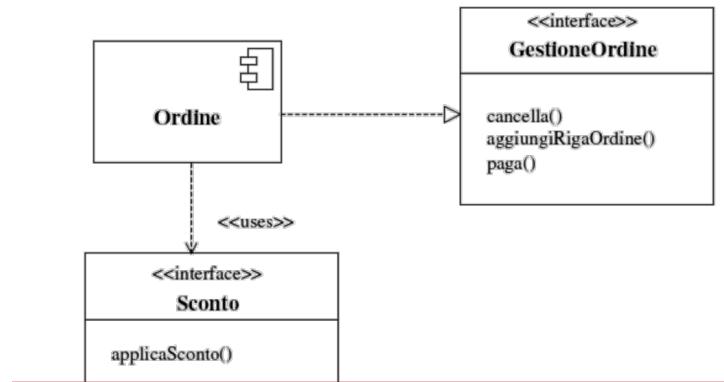
- Da UML 2 è una specializzazione della metaclassesse **Class**
- Quindi un componente può avere attributi e metodi, una struttura interna, porte e connettori
- Da UML 2 l'icona del componente è cambiata



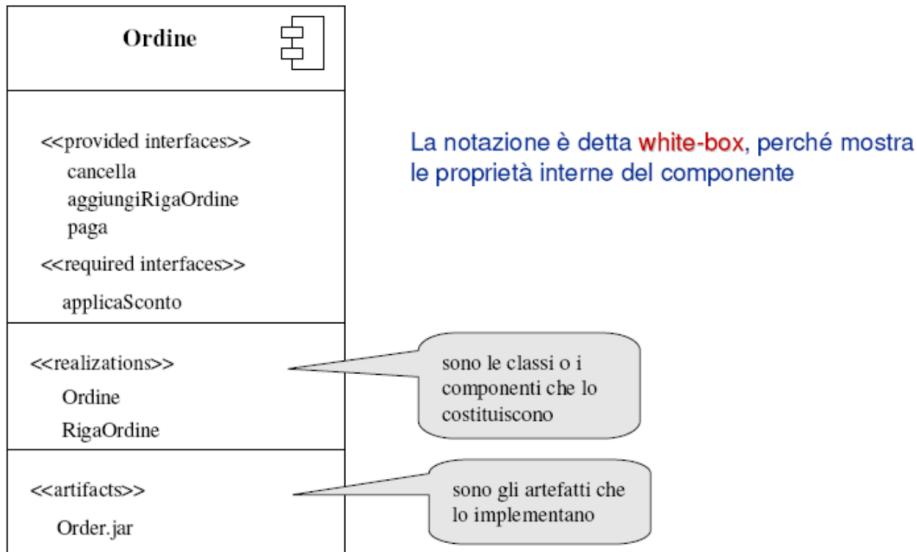
Relazioni tra componenti



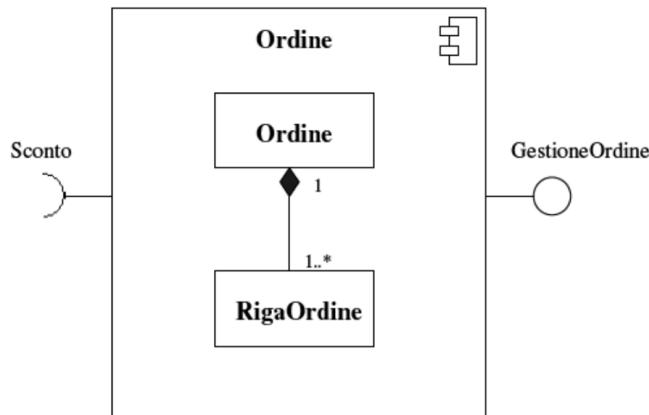
- Interfaccia fornita e interfaccia richiesta devono essere compatibili a livello di tipo (attributi e associazioni) e di vincoli sul comportamento (operazioni, eventi)



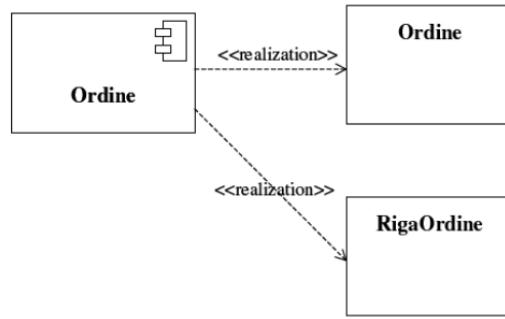
2.4.1.3 Componenti: White-box



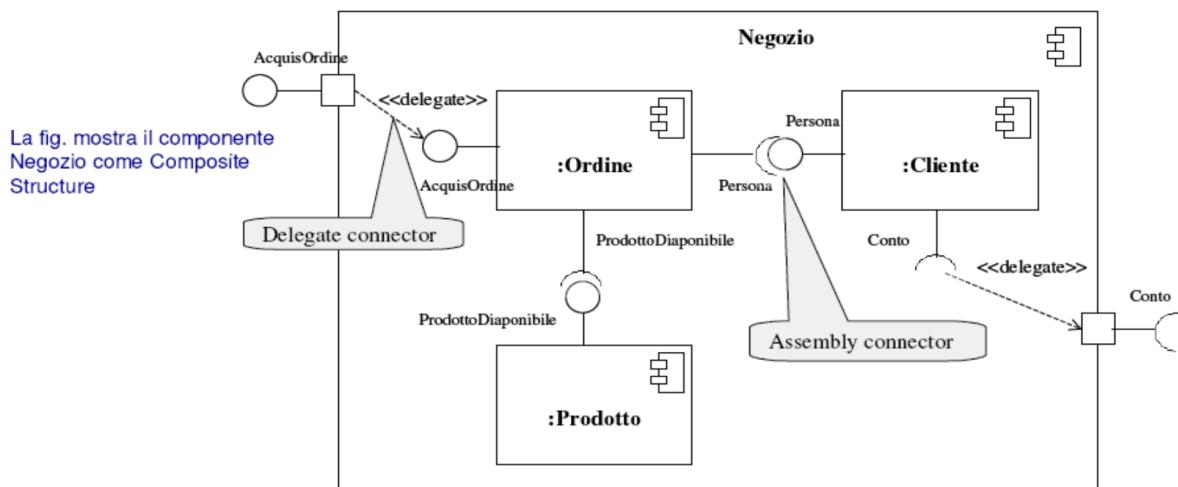
- I classificatori interni (internals) che realizzano un componente possono essere mostrati in due modi:
 - innestati nel componente



- In modo esplicito tramite la dipendenza di **realization**

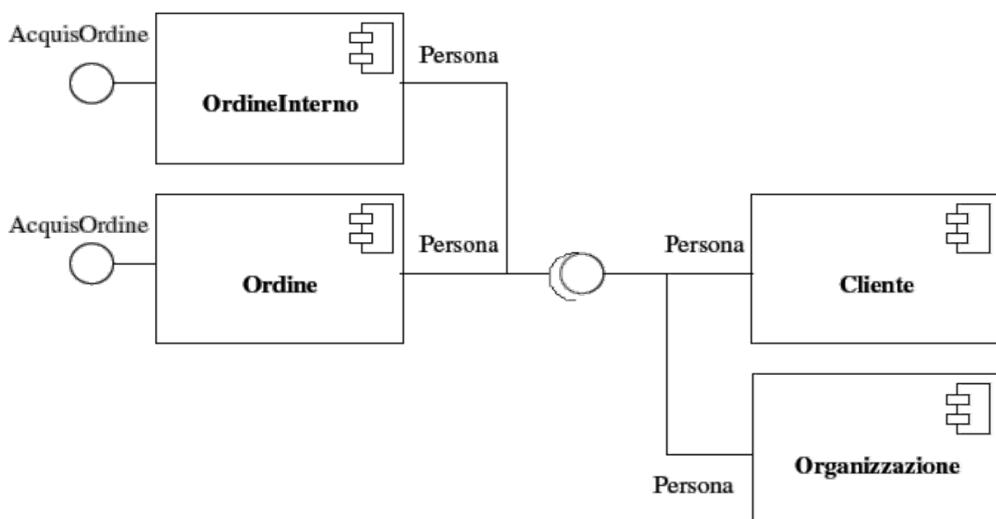


- Il componente *Ordine* è implementato istanziando le classi *Ordine* e *RigaOrdine*
- La **realization** è una dipendenza specializzata tra due insiemi di elementi di modellazione, di cui uno rappresenta la specifica e l'altro una sua implementazione
- Per un componente la **realization** definisce i classificatori che realizzano il contratto offerto dal componente stesso in termini delle sue interfacce offerte e richieste

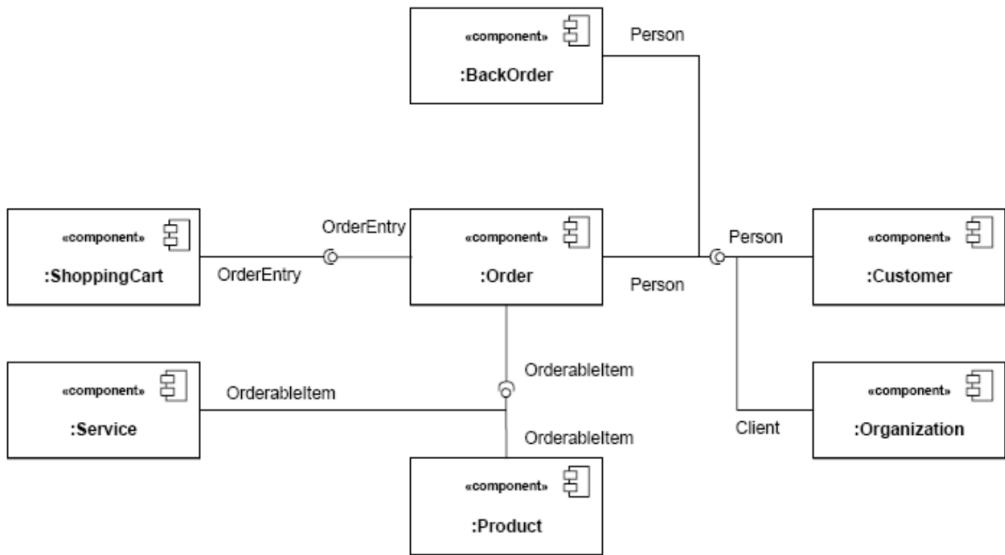


- Le parti interne sono collegate direttamente tra loro (**assembly connector**) oppure connesse a porte sul confine del componente (**delegated connector**)
- I delegated connector sono utilizzati per esporre servizi di una “parte” all'esterno del container

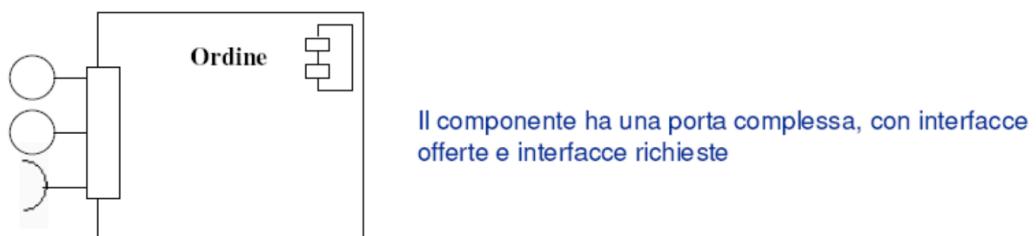
2.4.1.4 Connettori Multiple Wiring



- Entrambi i componenti, *Ordine* e *OrdineInterno*, richiedono l'interfaccia *Persona*: l'applicazione non conosce, fino al momento dell'esecuzione, quale componente, *Cliente* o *Organizzazione*, fornirà il servizio richiesto
- Si tratta di **un'interazione polimorfa**

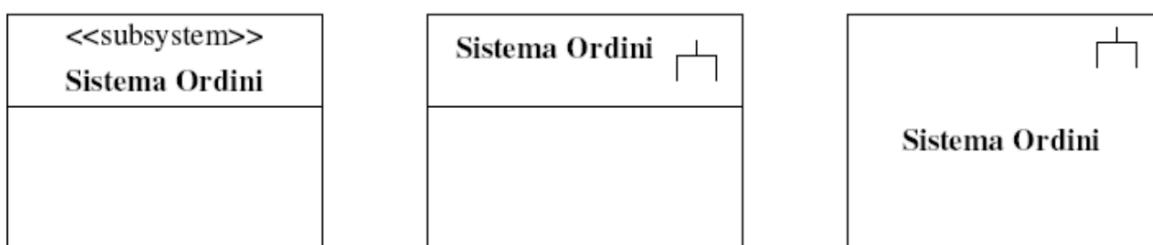


- UML 2.0 permette di connettere alla stessa porta più interfacce



2.4.1.5 Componenti e sottosistemi

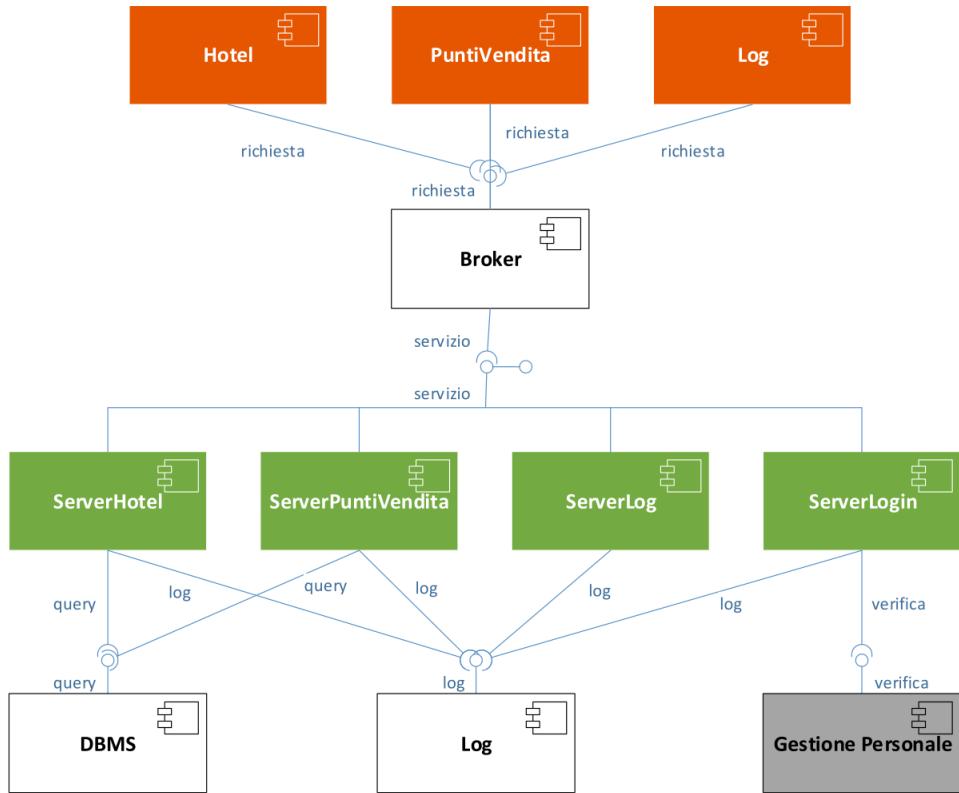
- Mentre in UML 1.x un **subsystem** è un tipo di package
- In UML 2 è un tipo di componente
- È quindi possibile specificare per un subsystem le interfacce richieste e quelle fornite, per evidenziare le relazioni con altri subsystem



2.4.1.6 Il diagramma

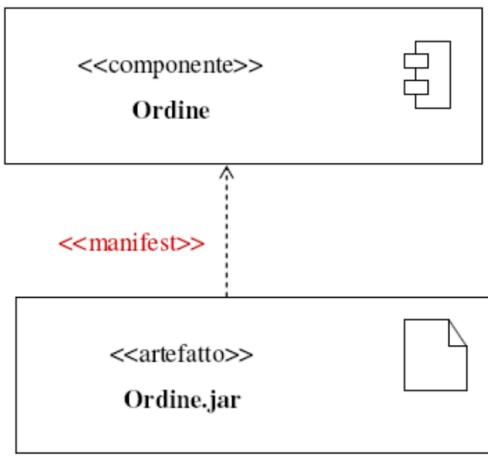
- Il diagramma dei componenti deve essere impiegato solamente negli stadi finali della fase di progettazione del sistema
- Tale diagramma rispecchia molto da vicino la struttura che dovrebbe avere il codice e, in un qualche modo, rappresenta l'architettura del sistema
- Potrebbe essere pensato come lo stadio finale dell'evoluzione dell'architettura logica che in fase di analisi viene rappresentata attraverso il diagramma dei package

2.4.1.7 Esempio Villaggio Turistico



2.4.2 Diagramma di Deployment

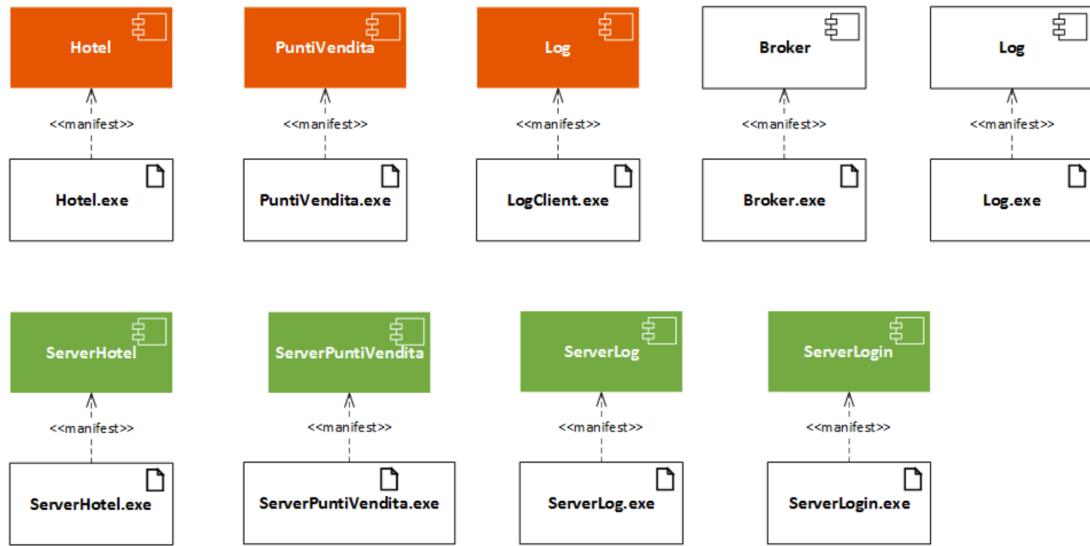
- I diagrammi di deployment **documentano la distribuzione fisica** di un sistema, mostrando i vari pezzi di software in esecuzione sulle macchine fisiche
- I diagrammi di deployment quindi mostrano:
 - i collegamenti che permettono la comunicazione fisica tra i pezzi hardware
 - le relazioni tra macchine fisiche e processi software, con l'indicazione dei vari punti in cui viene eseguito il codice
- Gli elementi principali del diagramma:
 - Artifact**
 - rappresenta una specifica porzione fisica di informazioni utilizzata o prodotta dal processo di sviluppo del software
 - esempi di artifact(manufatti): i modelli (un diagramma dei casi d'uso, un diagramma delle classi, ...), file sorgenti, script, file eseguibili, ...
 - tipicamente viene utilizzata una relazione di dipendenza <<manifest>> che illustra gli elementi di modellazione (in genere, componenti) utilizzati nella costruzione o generazione di un artefatto
 - Node**
 - un'unità sulla quali risiedono e/o sono eseguiti componenti/artefatti
 - i nodi comunicano tra loro tramite CommunicationPath
 - l'allocazione degli artefatti su un nodo viene rappresentata con una relazione di dipendenza <<deploy>> tra il nodo e l'artefatto
 - Device**
 - è una risorsa fisica computazionale con capacità elaborative sulla quale possono essere allocati artefatti per l'esecuzione
- “Manifest” è la relazione di dipendenza che illustra gli elementi di modellazione utilizzati nella costruzione o generazione di un artefatto



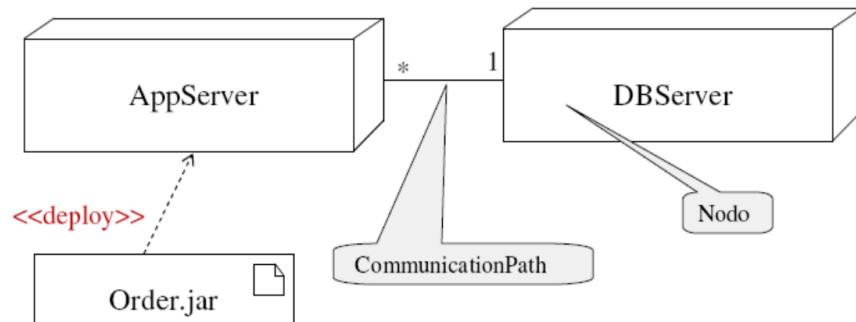
Il componente rappresenta un tipo di implementazione fisica, l'artefatto è l'attuale implementazione

In base ai principi di MDA, lo stesso tipo di componente può essere implementato in differenti tecnologie e quindi in differenti artefatti fisici

2.4.2.1 Esempio Villaggio Turistico

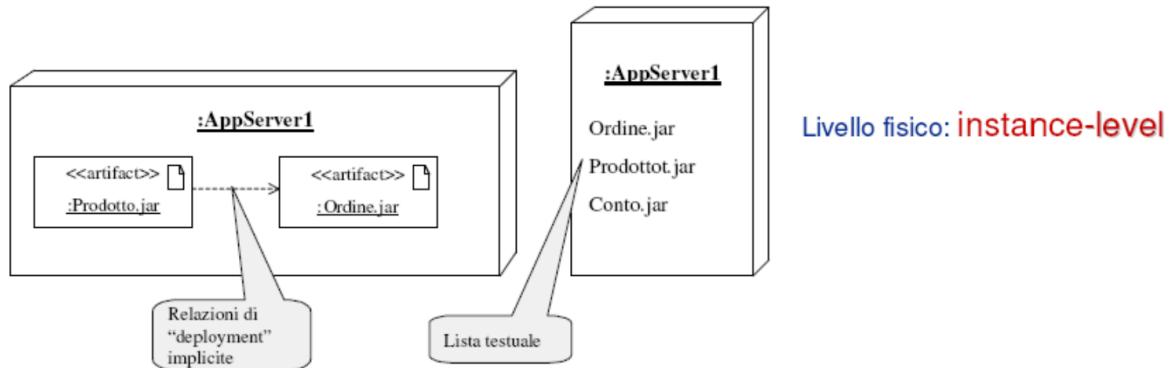
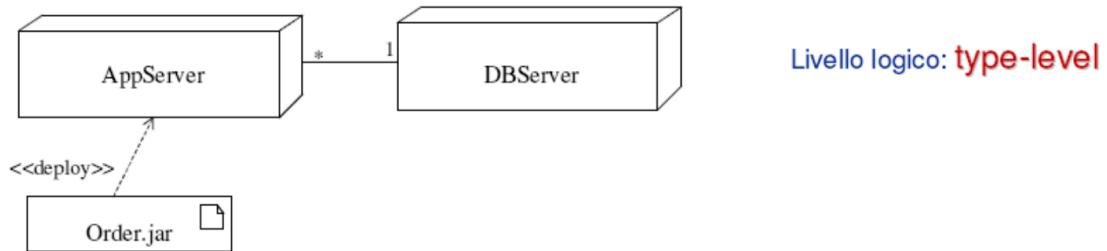


- Un **nodo** rappresenta qualsiasi cosa possa eseguire un lavoro: un server, un device o un'unità organizzativa
- È una risorsa su cui gli artefatti possono essere allocati per l'esecuzione, questo fatto viene rappresentato con una dipendenza di tipo <<deploy>> tra il nodo e l'artefatto



Un **communication path** è un'associazione tra due nodi tramite la quale i nodi possono scambiarsi segnali e messaggi

L'immagine ci dice che su ogni AppServer viene installato Order.jar, e ogni AppServer comunica con un DBServer.

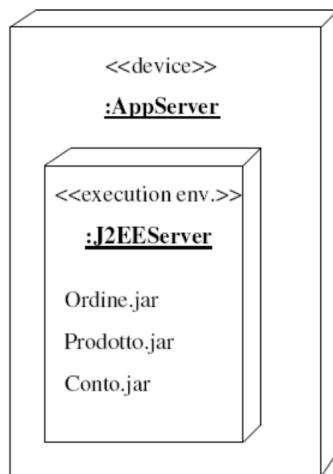


Tipicamente viene utilizzato il **type-level**, ma l'**instance-level** è più dettagliato.

- L'Execution Environment è un nodo che offre l'ambiente per l'esecuzione di specifici tipi di componenti che sono allocati su di esso

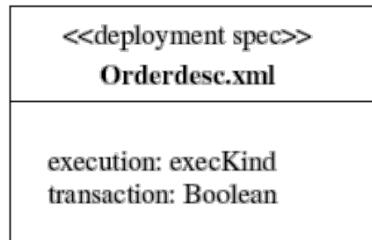
<<OS>>

- <<databasesystem>>
- <<J2EE container>>
- ...

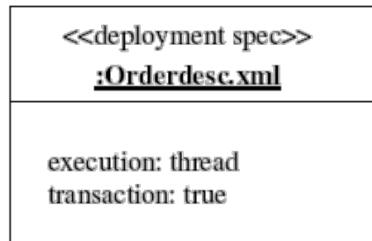


Con l'**instance-level** stiamo specificando che, per far funzionare un particolare componente, su quel dispositivo devono essere presenti tutti quegli artefatti.

- Deployment Specification: è un insieme di proprietà che determinano i parametri di esecuzione di un artefatto allocato su un nodo

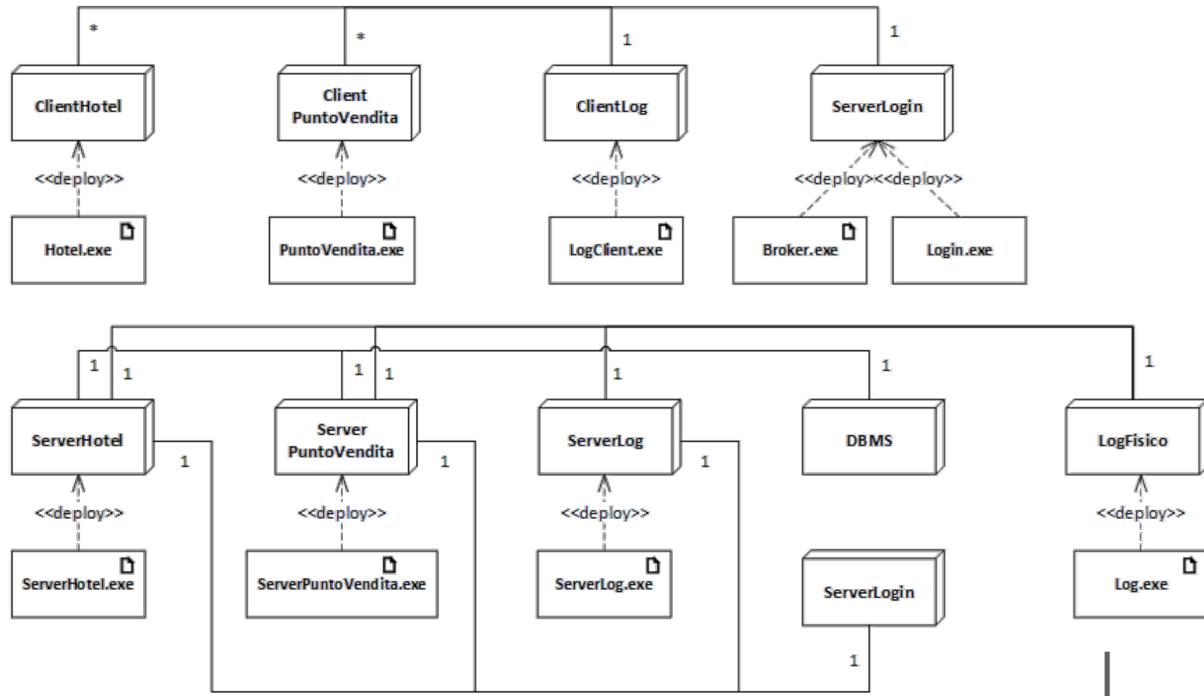


Type-level (Specification level)



Instance-level

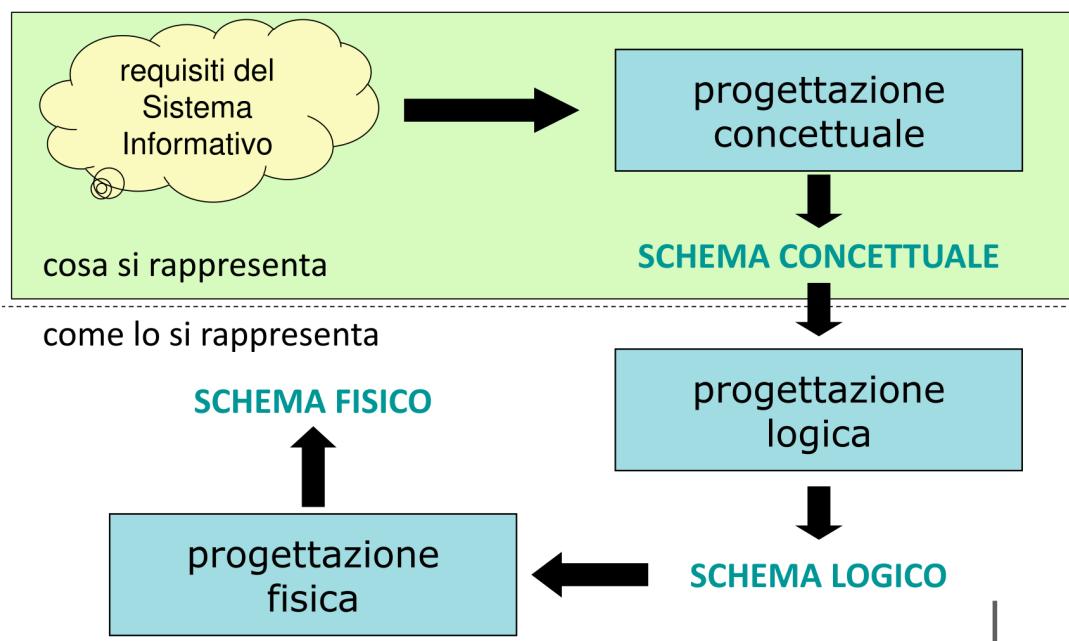
2.4.2.2 Esempio Villaggio Turistico



- In questo caso, `Broker.exe` e `Login.exe` devono essere presenti sulla stessa macchina `ServerLogin`
- C'è un solo `ClientLog`, e possono esserci tanti (indefiniti) `ClientPuntoVendita` e `ClientHotel`
- Nella parte sottostante non è necessario specificare il deployment del componente `ServerLogin`, perché già specificato sopra
- Il fatto che siano separati non implica che ci debbano essere due interfacce di rete diverse, ma solo che i percorsi sono diversi

Importante: in una applicazione web il deployment degli artefatti che manifestano un client non avviene sul client in sè, ma sul server; il client in questi casi è un semplice browser web.

2.5 Progettazione Concettuale (E/R)



2.5.1 Raccolta dei requisiti

- I requisiti devono innanzitutto essere acquisiti
- Le fonti possono essere molto diversificate tra loro:
 - **utenti**, attraverso:
 - interviste
 - documentazione apposita
 - **documentazione esistente**:
 - normative (leggi, regolamenti di settore)
 - regolamenti interni, procedure aziendali
 - realizzazioni preesistenti
 - **modulistica**
- La raccolta dei requisiti è un'attività difficile e non standardizzabile
 - in genere procede di pari passo con la fase di analisi (la prima analisi stimola nuove domande, ecc...)

2.5.2 Interagire con gli utenti

- È un'attività da considerare con molta attenzione, in quanto:
 - utenti diversi possono fornire informazioni diverse
 - utenti a livello più alto hanno spesso una visione più ampia ma meno dettagliata
- In generale, risulta utile:
 - effettuare spesso **verifiche** di comprensione e coerenza
 - verificare anche per mezzo di **esempi** (generali e relativi a casi limite)
 - richiedere **definizioni** e **classificazioni**
 - far evidenziare gli **aspetti essenziali** rispetto a quelli marginali

2.5.3 Requisiti: documentazione descrittiva

- Regole generali:
 - scegliere il corretto **livello di astrazione**
 - **standardizzare** la struttura delle frasi
 - **suddividere** le frasi articolate
 - **separare** le frasi sui **dati** da quelle sulle **funzioni** (operazioni)
- Per meglio evidenziare i concetti che sono espressi nei requisiti, è opportuno:
 - costruire un **glossario dei termini**

- individuare **omonimi** e **sinonimi** e unificare i termini
- rendere esplicito il **riferimento fra termini**
- riorganizzare le frasi per concetti

2.5.3.1 Esempio: BD bibliografica (1)

Si vogliono organizzare i dati di interesse per automatizzare la gestione dei riferimenti bibliografici, con tutte le informazioni da riportarsi in una bibliografia.

Per ogni pubblicazione deve esistere un codice identificante costituito da sette caratteri, indicanti le iniziali degli autori, l'anno di pubblicazione e un carattere aggiuntivo per la discriminazione delle collisioni (ad es. BL2007a)

- Dettagli marginali tendono solo a distrarre e non forniscono nessuna indicazione sulla struttura dello schema che si deve progettare

2.5.3.2 Esempio: BD bibliografica (2)

Si vogliono organizzare i dati di interesse per automatizzare la gestione dei riferimenti bibliografici, con tutte le informazioni da riportarsi in una bibliografia.

Le pubblicazioni sono di due tipi, monografie (per le quali interessano editore, data e luogo di pubblicazione) e articoli su rivista (con nome della rivista, volume, numero, pagine e anno di pubblicazione); per entrambi i tipi si debbono ovviamente riportare i nomi degli autori.

Per ogni pubblicazione deve esistere un codice identificante...

- Il paragrafo in grassetto fornisce informazioni utili per derivare lo schema concettuale, in quanto introduce concetti importanti nella realtà in esame

2.5.3.3 Un altro esempio più articolato

- Si vuole realizzare una base di dati per una società che eroga corsi, di cui vogliamo rappresentare i dati deiparticipantiai corsi e deidocenti.
- Per gli studenti(circa 5000), identificati da un codice, si vuole memorizzare il codice fiscale, il cognome, l'età, il sesso, il luogodi nascita, il nome dei loro attuali datori di lavoro, i posti dove hanno lavorato in precedenza insieme al periodo, l'indirizzo e il numero di telefono, icorsiche hanno frequentato (i corsi sono in tutto circa 200) e il giudizio finale.
- Rappresentiamo anche iseminariche stanno attualmente frequentando e, per ogni giorno, i luoghe le ore dove sono tenute le lezioni.
- I corsi hanno un codice, untitoloe possono avere varie edizioni con date di inizio e fine e numero di partecipanti.
- Se gli studenti sono liberi professionisti, vogliamo conoscere l'area di interesse e, se lo possiedono, iltitolo. Per quelli che lavorano alle dipendenze di altri, vogliamo conoscere invece il loro livello e la posizione ricoperta.
- Per gliinsegnanti(circa 300), rappresentiamo il cognome, l'età, ilpostodove sono nati, il nome del corso che insegnano, quelli che hanno insegnato nel passato e quelli che possono insegnare. Rappresentiamo anche tutti i loro recapiti telefonici. I docenti possono essere dipendenti interni della società o collaboratori esterni.

2.5.3.4 Glossario dei termini, omonimi e sinonimi

- Raramente i requisisti espressi in linguaggio naturale sono privi di ambiguità. È infatti frequente il caso di

Omonimi: lo stesso termine viene usato per descrivere concetti differenti (es: libro e copia di libro, posto: di lavoro e geografico)

Sinonimi: termini diversi vengono usati per descrivere lo stesso concetto (es: studente e partecipante)

- Un modo conveniente per rappresentare sinteticamente i concetti più rilevanti emersi dall'analisi è il glossario dei **termini**, il cui scopo è fornire per ogni concetto rilevante:
 - Una breve descrizione del concetto
 - Eventuali sinonimi
 - Relazioni con altri concetti del glossario stesso

2.5.3.4.1 Esempio

Termine	Descrizione	Sinonimi	Collegamenti
Partecipante	Persona che partecipa ai corsi. Può essere un dipendente o un professionista	Studente	Corso, Datore
Docente	Docente dei corsi. Può essere un collaboratore esterno	Insegnante	Corso
Corso	Corso organizzato dalla società. Può avere più edizioni	Seminario	Docente, Partecipante
Datore	Datori di lavoro attuali o passati dei partecipanti ai corsi	Posto	Partecipante

2.5.3.5 Ristrutturazione dei requisiti

- Oltre a costruire il glossario, al fine di semplificare le analisi successive, è utile riformulare i requisiti:
 - Eliminare le omonimie
 - Usare un termine univoco per ogni concetto
 - Riorganizzare le frasi raggruppandole in base al concetto cui si riferiscono
Nell'esempio:
 - Frasi di carattere generale
 - Frasi riferite ai partecipanti
 - Frasi riferite ai docenti
 - Frasi riferite ai corsi
 - Frasi riferite alle società

2.5.3.5.1 Esempio: frasi relative ai partecipanti

Per i partecipanti (circa 5000), identificati da un codice, rappresentiamo il codice fiscale, il cognome, l'età, il sesso, la città di nascita, i nomi dei loro attuali datori di lavoro e di quelli precedenti (insieme alle date di inizio e fine rapporto), le edizioni dei corsi che stanno attualmente frequentando e quelli che hanno frequentato nel passato, con la relativa votazione finale.

2.5.3.6 Dai concetti allo schema E/R

- Va sempre ricordato che un concetto non è di per sé un'entità, un'associazione, un attributo, o altro **DIPENDE DAL CONTESTO!**
- Come regole guida, un concetto verrà rappresentato come

- ▶ Entità
 - se ha proprietà significative e descrive oggetti con esistenza autonoma
- ▶ Attributo
 - se è semplice e non ha proprietà
- ▶ Associazione
 - se correla due o più concetti
- ▶ Generalizzazione/specializzazione
 - se è caso più generale/particolare di un altro

2.5.3.7 Strategie di progettazione

- Per affrontare progetti complessi è opportuno adottare uno specifico modo di procedere, ovvero una **strategia di progettazione**
- I casi notevoli sono:
 - ▶ Strategia **top-down**:
Si parte da uno schema iniziale molto astratto ma completo, che viene successivamente raffinato fino ad arrivare allo schema finale
 - ▶ Strategia **bottom-up**:
Si suddividono le specifiche in modo da sviluppare semplici schemi parziali ma dettagliati, che poi vengono integrati tra loro
 - ▶ Strategia **inside-out**:
Lo schema si sviluppa “a macchia d’olio”, partendo dai concetti più importanti, che quindi vengono espansi aggiungendo quelli a essi correlati, e così via

2.5.3.7.1 Pro e contro delle strategie

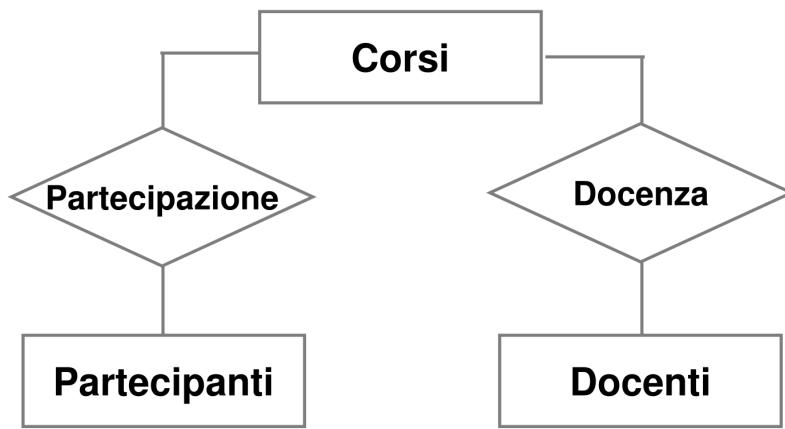
Strategia	Pro	Contro
Top-down	non è inizialmente necessario specificare i dettagli	richiede sin dall'inizio una visione globale del problema , non sempre ottenibile in casi complessi
Bottom-up	permette una ripartizione delle attività	richiede una fase di integrazione
Inside-out	non richiede passi di integrazione	richiede ad ogni passo di esaminare tutte le specifiche per trovare i concetti non ancora rappresentati

2.5.3.7.2 Un approccio “misto”

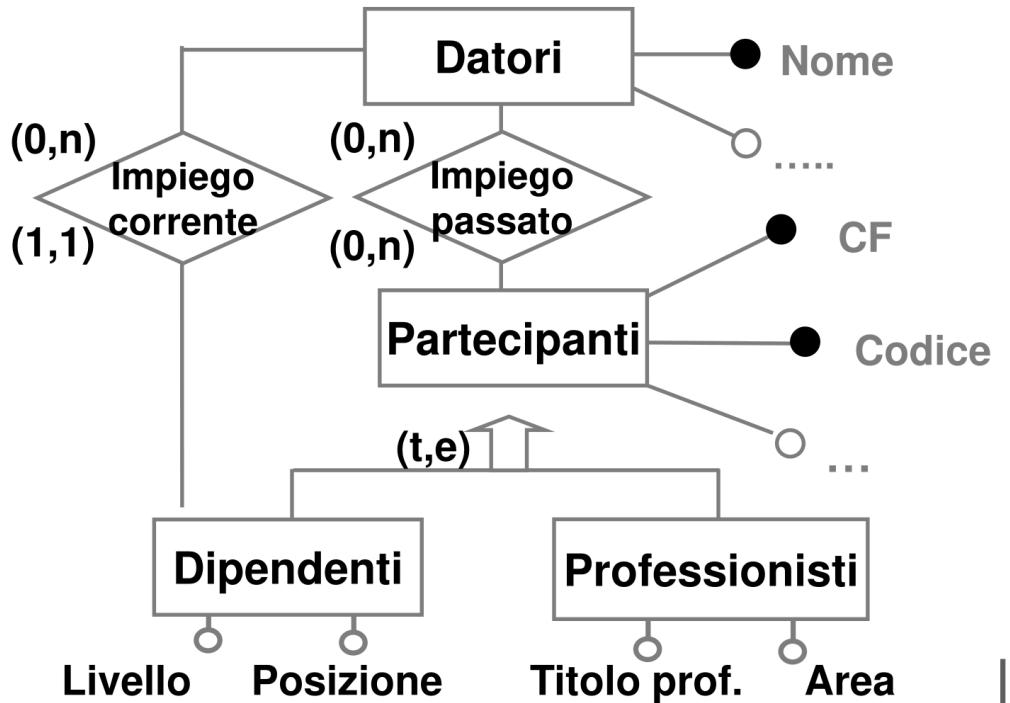
- Nella pratica si fa spesso uso di una strategia ibrida, nella quale:
 1. si individuano i concetti principali e si realizza uno **schema scheletro**, che contiene solamente i concetti più importanti
 2. sulla base di questo si può **decomporre**
 3. poi si raffina, si espande, si integra

... vediamo cosa succede nel caso della società di formazione...

2.5.3.7.3 Società di formazione: schema scheletro

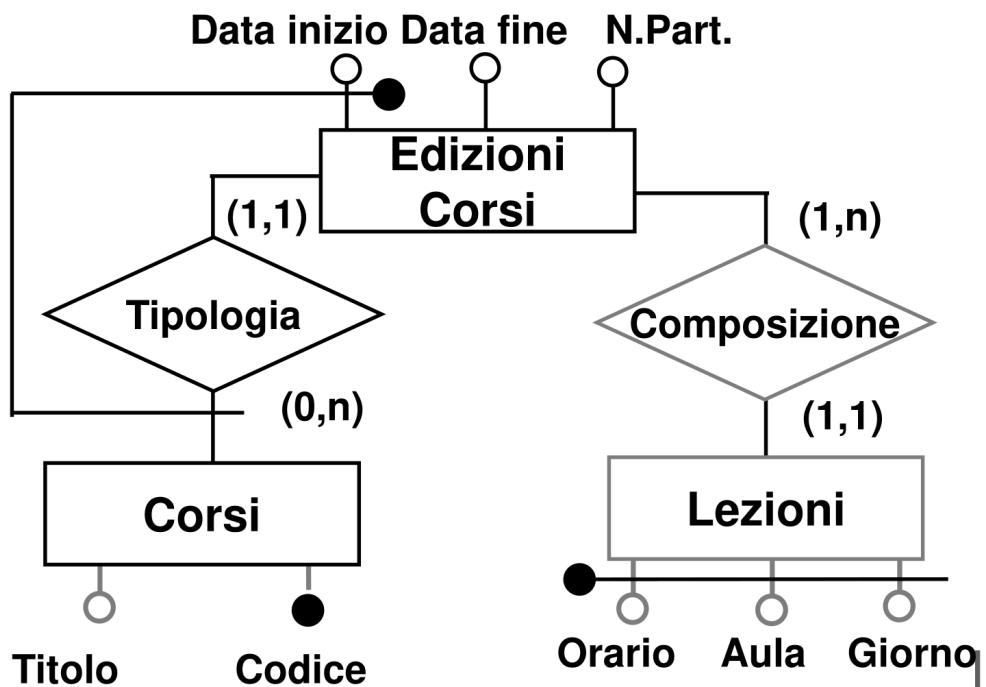


Raffinamento di Partecipanti



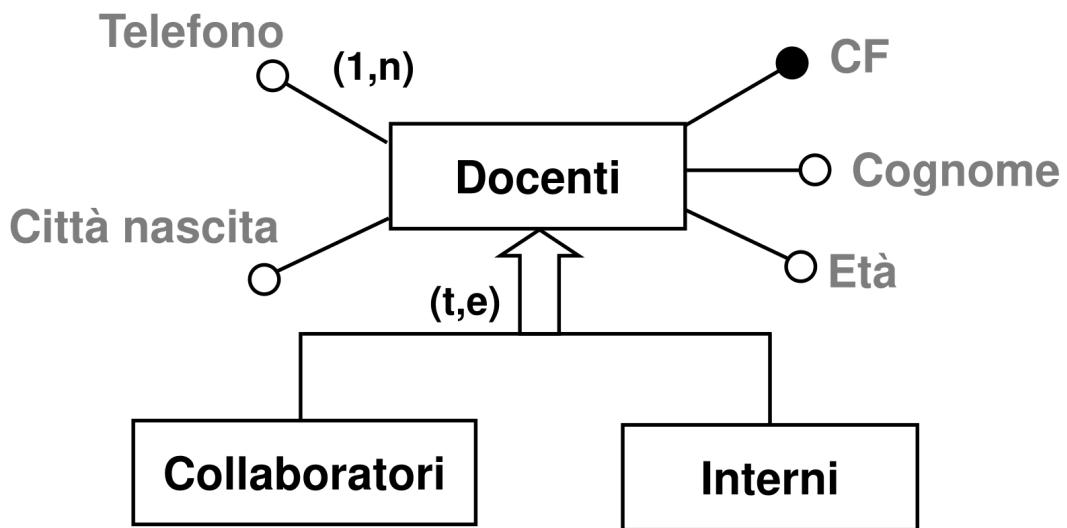
- Dato che i partecipanti possono essere o dipendenti o professionisti, facciamo una gerarchie totale ed esclusiva (t,e)
- I dipendenti possono avere (impiego corrente) un unico datore di lavoro, mentre i datori possono avere da 0 a n dipendenti
- Impiego passato si riferisce alla classe generale Partecipanti perché uno che adesso è un Professionista può essere stato, in passato, un Dipendente

Raffinamento di Corsi

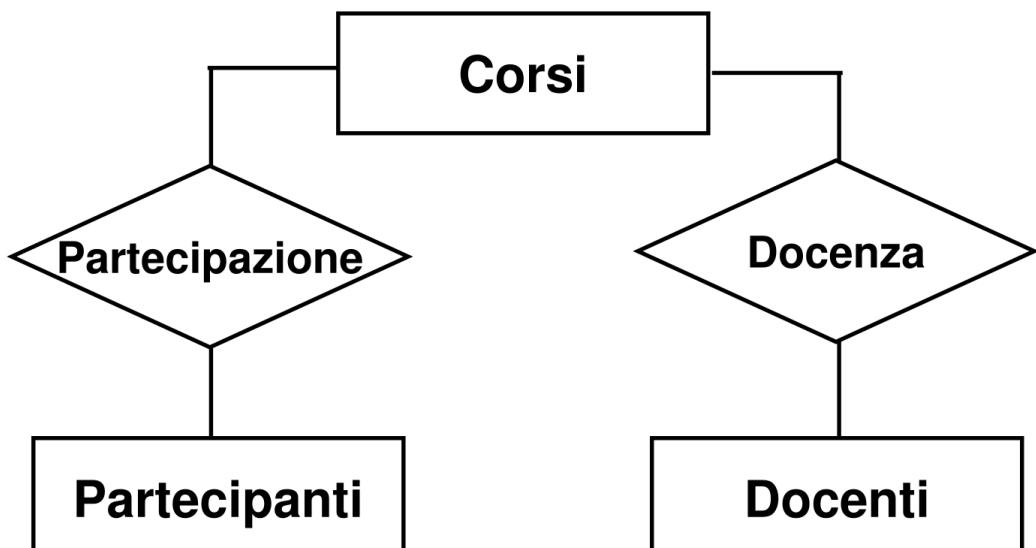


- La chiave di Edizioni Corsi è il Codice del corso e la Data inizio

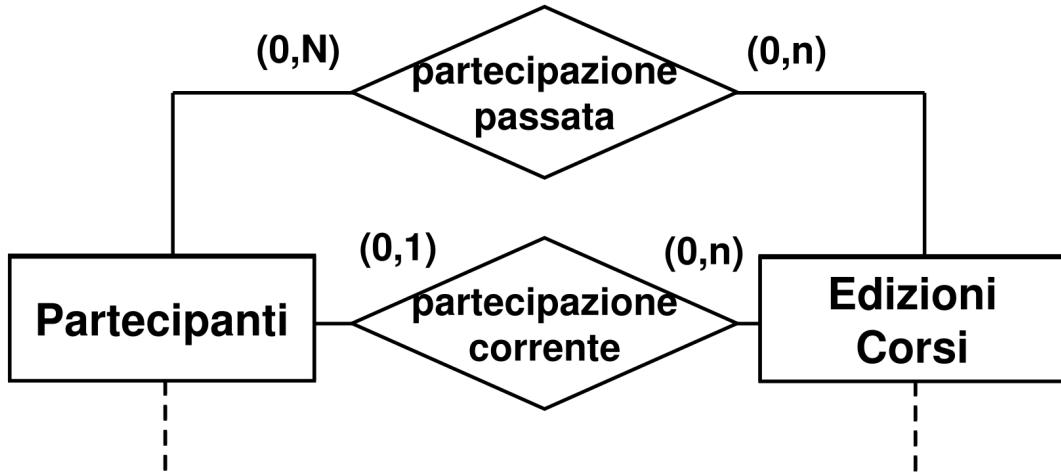
Raffinamento di Docenti



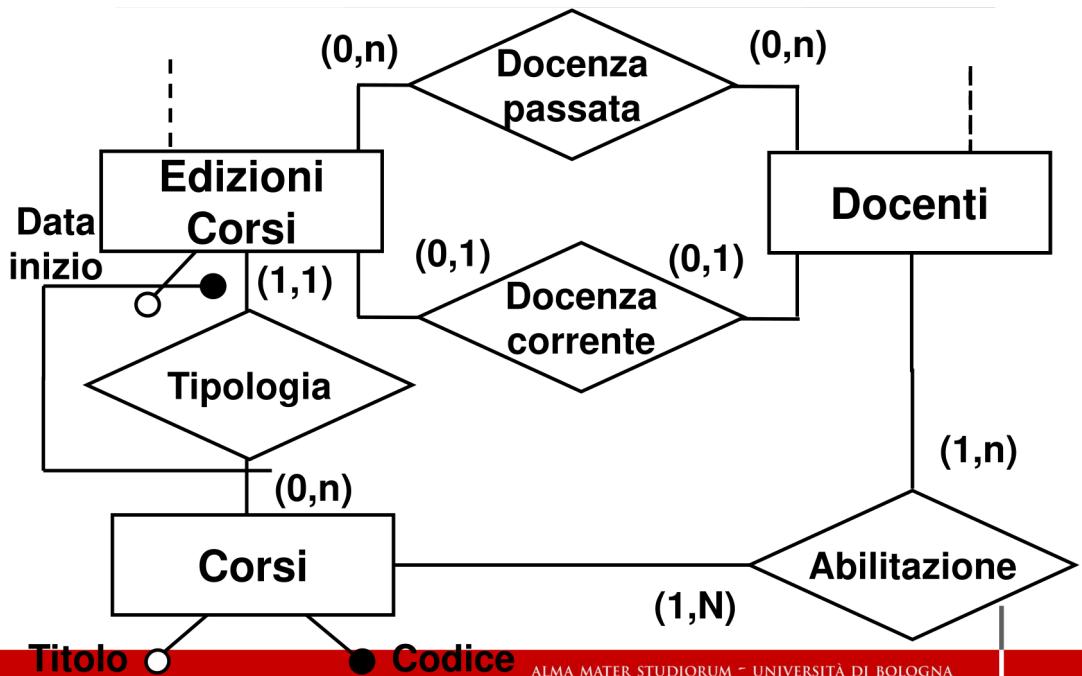
Integrazione: schema di riferimento



Integrazione: Partecipanti e Corsi



Integrazione: Docenti e Corsi



- C'è un vincolo non espresso sulla data di Docenza corrente e Docenza passata

2.5.3.8 Qualità di uno schema concettuale

- Lo schema E/R deve essere verificato** accuratamente per verificare che risponda a requisiti di:
 - Correttezza**
 - Non devono essere presenti errori (sintattici o semantici)
 - Completezza**
 - Tutti i dati di interesse devono essere specificati
 - Leggibilità**
 - Riguarda anche aspetti prettamente estetici dello schema
 - Minimalità**
 - È importante capire se esistono elementi ridondanti nello schema; in alcuni casi ciò non è un problema, ma può essere viceversa una scelta di progettazione volta a favorire l'esecuzione di certe operazioni

2.5.3.9 Metodologia basata sulla strategia mista

Analisi dei requisiti

- Analizzare i requisiti ed eliminare le ambiguità
- Costruire un glossario dei termini, raggruppare i requisiti

Passo base

- Definire uno schema scheletro con i concetti più rilevanti

Passo di decomposizione (se necessario o appropriato)

- Decomporre i requisiti con riferimento ai concetti nello schema scheletro

Passo iterativo (da ripetere finché non si è soddisfatti)

- Raffinare i concetti presenti sulla base delle loro specifiche
- Aggiungere concetti per descrivere specifiche non descritte

Passo di integrazione (se si è decomposto)

- Integrare i vari sottoschemi in uno schema complessivo, facendo riferimento allo schema scheletro

Analisi di qualità (ripetuta e distribuita)

- Verificare le qualità dello schema e modificarlo

2.5.3.10 Riassumendo

- La **fase di analisi dei requisiti** è fondamentale per poter progettare una base di dati che rispetti i requisiti
- Mancando la possibilità di standardizzarla, tale fase si avvale necessariamente di regole di buon senso e di una serie di strumenti che riducono il rischio di commettere errori grossolani, oltre a costituire una valida documentazione
- Per la **progettazione dello schema E/R** sono possibili diverse strategie, di cui quella **mista** è senz'altro la più diffusa e adeguata anche nel caso di progetti estremamente complessi

3 Framework .NET

3.1 Introduzione

3.1.1 Tecnologia COM - Component Object Model

- Nasce nel 1993
- COM è un sistema platform-independent, distribuito, object-oriented per la creazione di componenti software binari
- COM non è un linguaggio object-oriented ma uno **standard**
- COM specifica un modello a oggetti e requisiti di programmazione che permettono agli oggetti COM di interagire con altri oggetti
- Ad esempio un programma Java può parlare con un programma C, utilizzando la Java Native Interface
- COM fa la stessa cosa del linker C quando abbiamo bisogno che più programmi parlino tra loro (link dinamico)

Esempio di oggetti COM che parlano tra di loro:

```
interface IUnknown
{
    virtual HRESULT QueryInterface(IID iid, void **ppvObject) = 0;
```

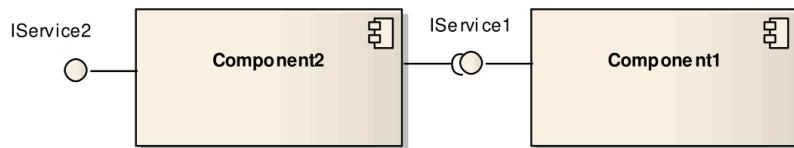
```

    virtual ULONG AddRef(void) = 0;
    virtual ULONG Release(void) = 0;
};

```

- **QueryInterface** è utilizzata per ottenere un puntatore a un'altra interfaccia (a tempo di esecuzione, non di compilazione), dato un GUID che identifica univocamente tale interfaccia (noto comunemente come interfaceID, o IID)
 - se l'oggetto COM non implementa tale interfaccia, viene restituito un errore **E_NOINTERFACE**
- **AddRef** è utilizzato dai client per indicare che un oggetto COM viene referenziato (usato)
- **Release** è utilizzato dai client per indicare che hanno finito di utilizzare l'oggetto COM
- Le specifiche COM richiedono l'utilizzo di una tecnica chiamata **reference counting** per assicurare che i singoli oggetti rimangano “vivi” fintantoché esistono client che hanno ottenuto l'accesso a una o più delle loro interfacce e, di converso, che gli stessi oggetti vengano appropriatamente cancellati quando i clienti che li utilizzavano hanno finito di usarli e non ne hanno più bisogno; il contatore viene incrementato a ogni **AddRef** e decrementato a ogni **Release**
- Un oggetto COM è responsabile della liberazione della propria memoria una volta che il suo reference count arrivi a zero
- Il reference counting può causare **problemi se due o più oggetti hanno riferimenti circolari**
- Ereditarietà solo con composizione e delega

interface IService2 : IService1



- La **posizione** di ciascun componente è memorizzata nel registro Windows
- Di un certo componente può esistere un'unica versione installata
- Questa limitazione può complicare seriamente

il deployment di applicazioni basate su COM, a causa della possibilità che diversi programmi, o anche diverse versioni dello stesso programma, siano progettati per funzionare con versioni diverse dello stesso componente COM

- Questa situazione è nota anche come inferno delle DLL (DLL hell)

3.1.2 Cos'è il Framework .NET

- Ambiente di esecuzione (runtime environment) + Libreria di classi (standard + estensioni MS)
- Versione 1.0 del 2002 ► v. 4.8 (versione definitiva, 7/19)
- Semplifica lo sviluppo e il deployment

- Aumenta l'affidabilità del codice
- Unifica il modello di programmazione
- È completamente indipendente da COM
- È fortemente integrato con COM
- Ambiente object-oriented
 - Qualsiasi entità è un oggetto
 - Classi ed ereditarietà pienamente supportati
- Riduzione errori comuni di programmazione
 - **Garbage Collector**
 - Linguaggi fortemente tipizzati - **Type Checker**
 - Errori non gestiti ▷ generazione di eccezioni
- Libertà di scelta del linguaggio
 - Funzionalità del framework disponibili in tutti i linguaggi .NET
 - I componenti della stessa applicazione possono essere scritti in linguaggi diversi
 - Ereditarietà supportata anche tra linguaggi diversi
- Possibilità di estendere una qualsiasi classe .NET (non sealed) mediante ereditarietà
- Diversamente da COM:
 - si usa e si estende la classe stessa
 - non si deve utilizzare composizione e delega
- .NET è un'implementazione di CLI
 - Common Language Infrastructure
- CLI e il linguaggio C# sono standard ECMA
 - ECMA-334 (C#), ECMA-335 (CLI)
- Esistono altre implementazioni di CLI:
 - SSCLI (Shared Source CLI by Microsoft, per Windows, FreeBSD e Macintosh) - Rotor
 - Mono (per Linux)
 - DotGNU
 - Intel OCL (Open CLI Library)
 - ...

3.1.3 Standard ECMA-335

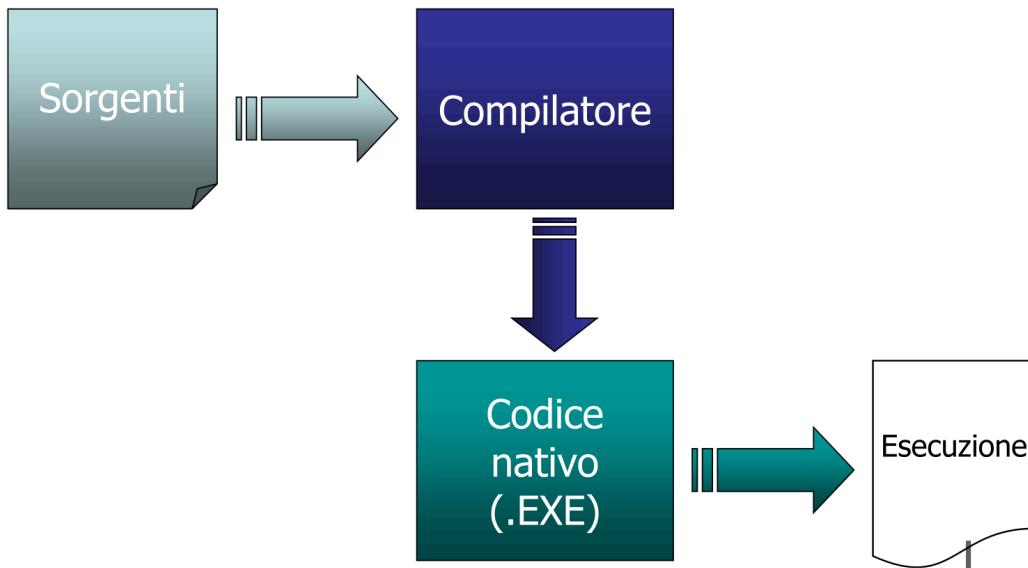
- Definisce la Common Language Infrastructure (CLI) nella quale applicazioni scritte in **diversi linguaggi di alto livello** possono essere eseguite in **diversi ambienti di sistema** senza la necessità di riscrivere l'applicazione per prendere in considerazione le caratteristiche peculiari di tali ambienti
- CLI è un **ambiente a tempo di esecuzione**, con:
 - un formato di file
 - un sistema di tipi comune
 - un sistema di metadati estensibile
 - un linguaggio intermedio
 - accesso alla piattaforma sottostante
 - una libreria di classi base

- Concetti chiave:
 - **(Microsoft) Intermediate Language** - (MS)IL
 - **Common Language Runtime** - CLR
 - ambiente di esecuzione runtime per le applicazioni .NET
 - il codice che viene eseguito sotto il suo controllo si dice **codice gestito** (managed)
 - **Common Type System** - CTS
 - tipi di dato supportati dal framework .NET
 - consente di fornire un modello di programmazione unificato
 - **Common Language Specification** - CLS
 - regole che i linguaggi di programmazione devono seguire per essere interoperabili all'interno del framework .NET
 - sottoinsieme di CTS

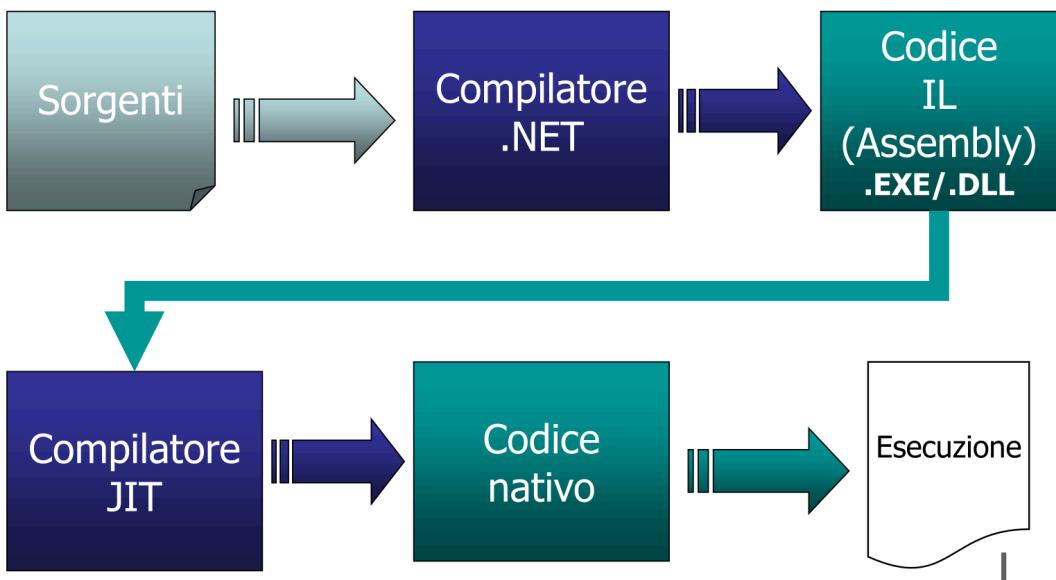
3.1.4 Codice interpretato



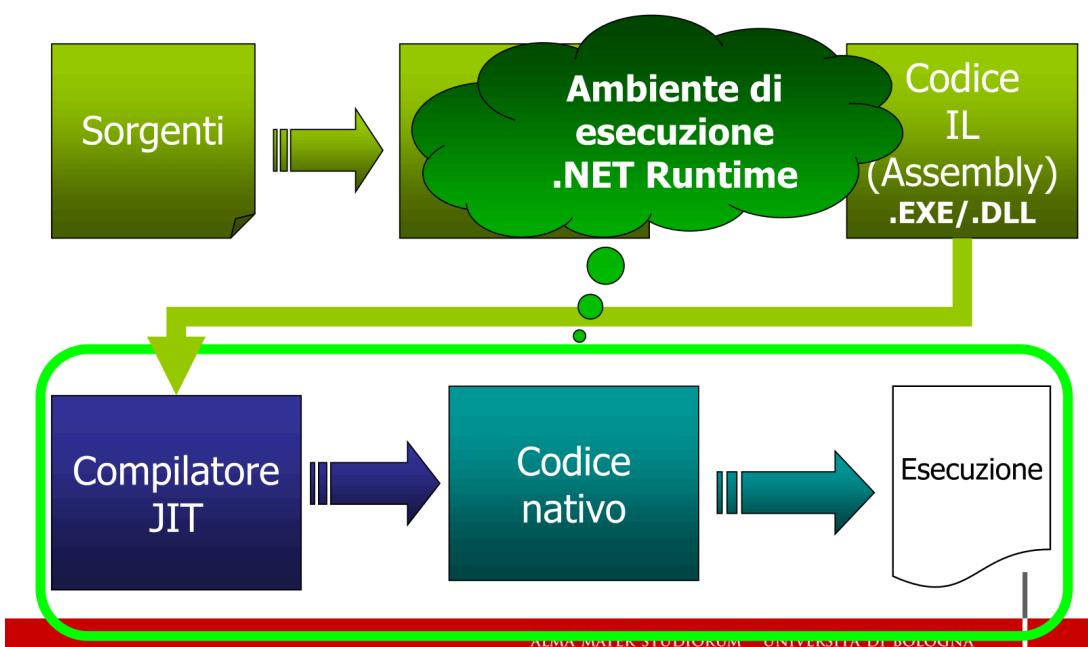
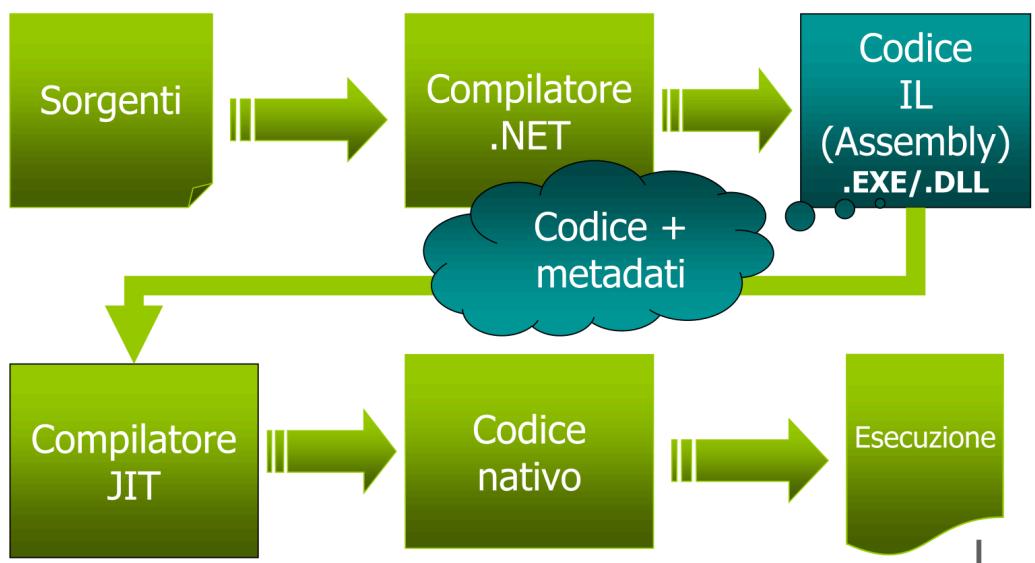
3.1.5 Codice nativo



3.1.6 Codice IL

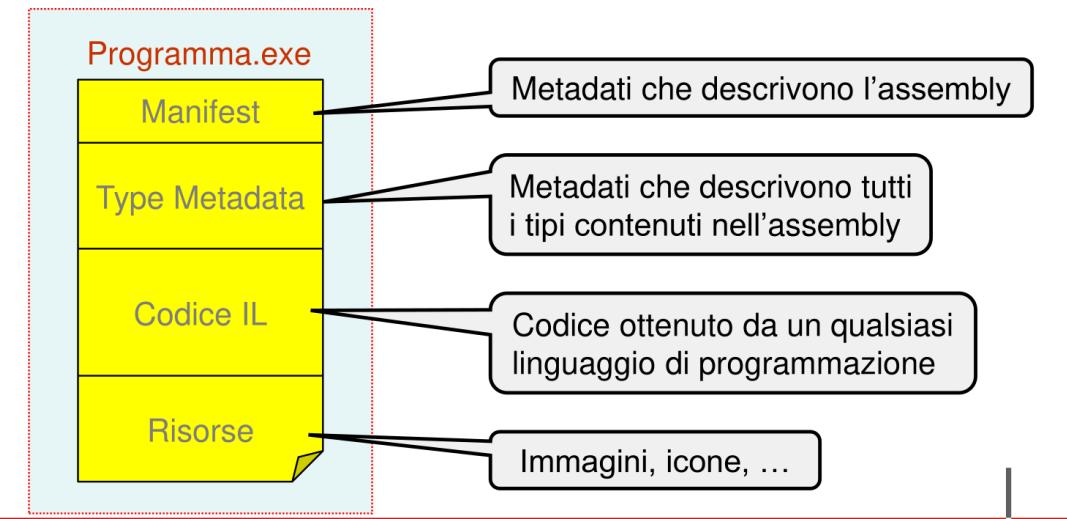


JIT: just in time

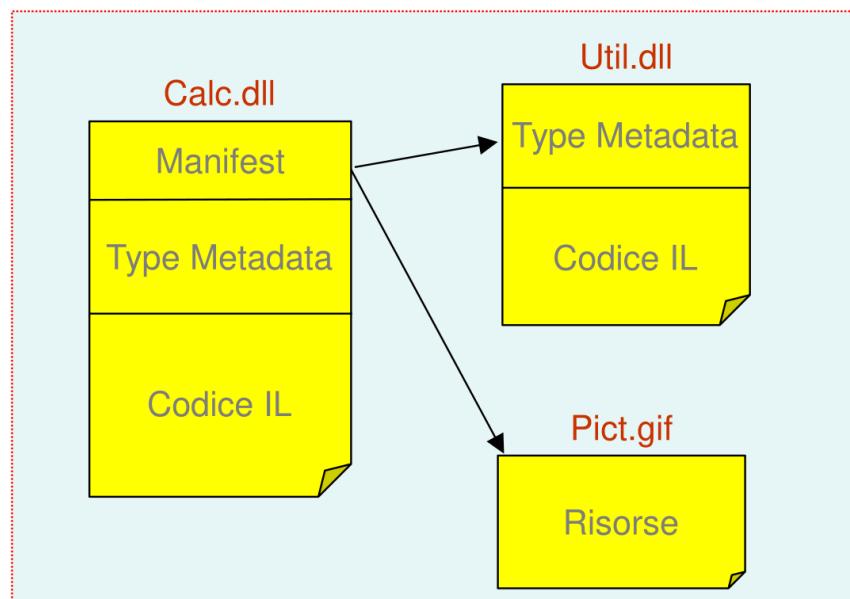


3.1.7 Assembly

- Unità minima per la distribuzione e il versioning
- Normalmente è composto da un solo file



- Ma può essere composto anche da più file



3.1.8 Metadati

- Descrizione dell'assembly - Manifest
 - Identità: nome, versione, cultura [, public key]
 - Lista dei file che compongono l'assembly
 - Riferimenti ad altri assembly da cui si dipende
 - Permessi necessari per l'esecuzione
 - ...
- Descrizione dei tipi contenuti nell'assembly
 - Nome, visibilità, classe base, interfacce
 - Campi (attributi membro), metodi, proprietà, eventi, ...
 - Attributi (caratteristiche aggiuntive)
 - definiti dal compilatore
 - definiti dal framework
 - definiti dall'utente

3.1.9 Chi usa i metadati?

- Compilatori
 - Compilazione condizionale
- Ambienti RAD (Rapid Application Development)
 - Informazioni sulle proprietà dei componenti

- Categoria
- Descrizione
- Editor specializzati per tipo di proprietà
- Tool di analisi dei tipi e del codice
 - Intellisense, ILDASM, Reflector, ...
- Sviluppatori - **Reflection** (introspezione)
 - Analisi del contenuto di un assembly, permetter agli altri di guardare dentro

3.1.10 Esempio Assembly

```
.assembly Hello { }
.assembly extern mscorel { }
.method public static void main()
{
    .entrypoint
    ldstr "Hello IL World!"
    call void [mscorel]System.Console::WriteLine
    (class System.String)
    ret
}

ilasm helloil.il
```

3.1.11 Dove trovare gli Assembly

- **Assembly privati**
 - Utilizzati da un'applicazione specifica
 - Directory applicazione (e sub-directory)
- **Assembly condivisi**
 - Utilizzati da più applicazioni
 - Global Assembly Cache (GAC)
 - c:\windows\assembly
- **Assembly scaricati da URL**
 - Download cache
 - c:\windows\assembly\download

3.1.12 Deployment semplificato

- Installazione senza effetti collaterali
 - Applicazioni e componenti possono essere
 - condivisi
 - privati
- Esecuzione side-by-side
 - Versioni diverse dello stesso componente possono coesistere, anche nello stesso processo

3.1.13 Common Language Runtime

VB

C++

C#

JScript

...

Common Language Specification

Web
Services

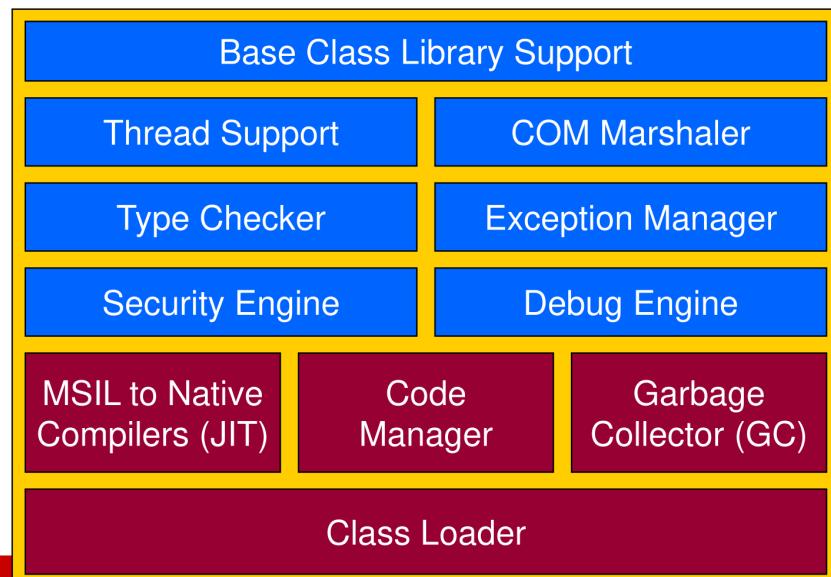
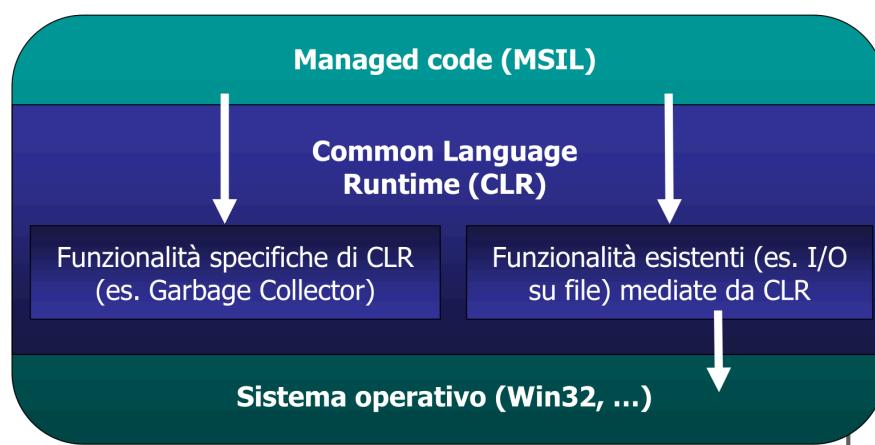
User
Interface

Data and XML

Base Class Library

Common Language Runtime

- IL CLR offre vari servizi alle applicazioni



3.1.14 Garbage Collector

- Gestisce il ciclo di vita di tutti gli oggetti .NET
- Gli oggetti vengono distrutti automaticamente quando non sono più referenziati
- A differenza di COM, non si basa sul Reference Counting

- ▶ Maggiore velocità di allocazione
- ▶ Consentiti i riferimenti circolari
- ▶ Perdita della distruzione deterministica: non posso sapere in che momento un determinato oggetto verrà distrutto
- ▶ Inoltre, essendo più complicato del garbage collector di COM, l'esecuzione dello stesso è un po' più pesante

3.1.15 Gestione delle eccezioni

- Praticamente uguali a quelle di Java
- Un'eccezione è
 - ▶ una condizione di errore
 - ▶ un comportamento inaspettato

incontrato durante l'esecuzione del programma

- Un'eccezione può essere generata da
 - ▶ codice del programma in esecuzione
 - ▶ ambiente di runtime
- In CLR, un'eccezione è un oggetto che eredita dalla classe `System.Exception`
- Gestione uniforme, elimina
 - ▶ codici HRESULT di COM
 - ▶ codici di errore Win32
 - ▶ ...
- Concetti universali
 - ▶ Lanciare un'eccezione (`throw`)
 - ▶ Catturare un'eccezione (`catch`)
 - ▶ Eseguire codice di uscita da un blocco controllato (`finally`)
- Disponibile in tutti i linguaggi .NET con sintassi diverse

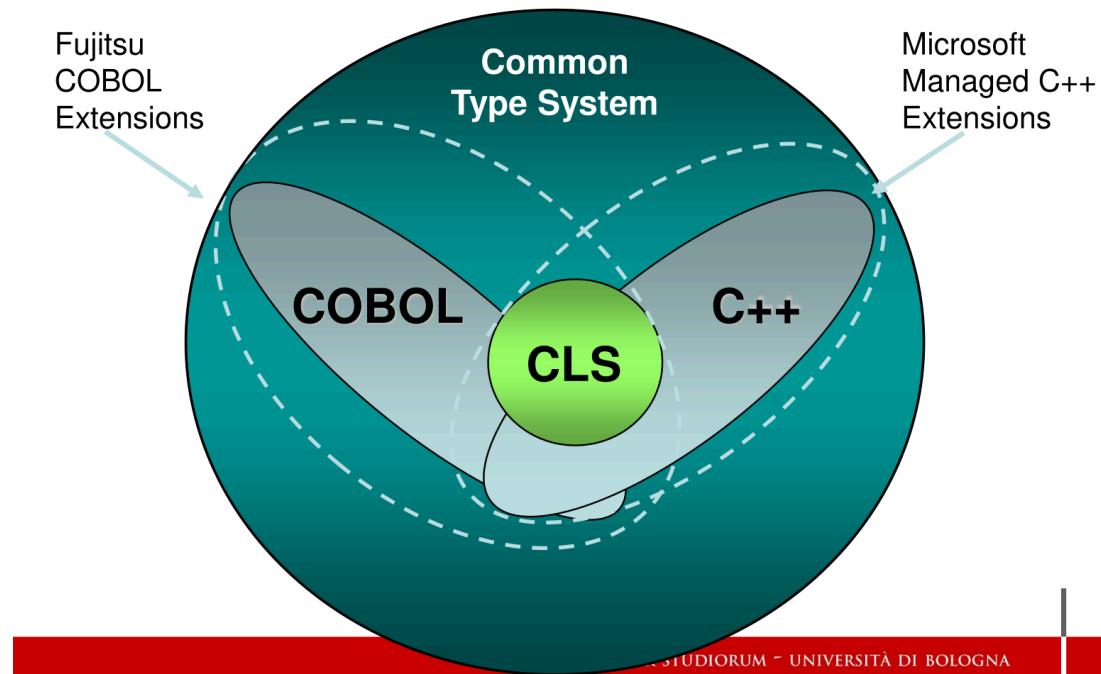
3.1.16 Common Type System

- Tipi di dato supportati dal framework .NET
 - ▶ Alla base di tutti i linguaggi .NET
- Fornisce un modello di programmazione unificato
- Progettato per linguaggi object-oriented, procedurali e funzionali
 - ▶ Esaminate caratteristiche di 20 linguaggi
 - ▶ Tutte le funzionalità disponibili con IL
 - ▶ Ogni linguaggio utilizza alcune caratteristiche
- Alla base di tutto ci sono i tipi: **classi, strutture, interfacce, enumerativi, delegati**
- Fortemente tipizzato (compile-time)
- Object-oriented
 - ▶ Campi, metodi, tipi annidati, proprietà, ...
- Overload di metodi (compile-time)
- Invocazione metodi virtuali risolta a run-time

- Ereditarietà singola di estensione
- Ereditarietà multipla di interfaccia

3.1.17 Common Language Specification

- Definisce le regole di compatibilità tra linguaggi (sottoinsieme di CTS)
 - Regole per gli identificatori
 - Unicode, case-sensitivity
 - Keyword
 - Regole per denominazione proprietà ed eventi
 - Regole per costruttori degli oggetti
 - Regole di overload più restrittive
 - Ammesse interfacce multiple con metodi con lo stesso nome
 - Non ammessi puntatori unmanaged
 - ...



3.1.17.1 Tipi nativi

CTS	C#
System.Object	object
System.String	string
System.Boolean	bool
System.Char	char
System.Single	float
System.Double	double
System.Decimal	decimal
System.SByte	sbyte
System.Byte	byte
System.Int16	short
System.UInt16	ushort
System.Int32	int
System.UInt32	uint
System.Int64	long
System.UInt64	ulong

3.1.18 Common Type System

- Tutto è un oggetto
 - `System.Object` è la classe radice
- Due categorie di tipi
 - Tipi riferimento
 - Riferimenti a oggetti allocati sull'**heap** gestito
 - Indirizzi di memoria
 - Tipi valore
 - Allocati sullo **stack** o parte di altri oggetti
 - Sequenza di byte
- Sono memorizzati come in Java

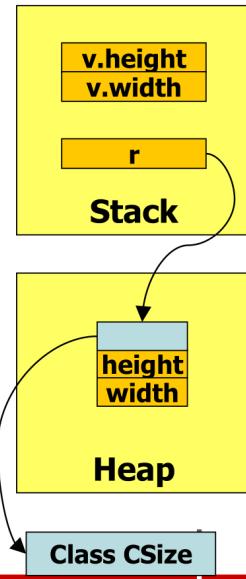
3.1.18.1 Tipi valore

- I tipi valore comprendono:
 - Tipi primitivi (built-in)
 - `Int32`, ...
 - `Single`, `Double`
 - `Decimal`
 - `Boolean`
 - `Char`
 - Tipi definiti dall'utente
 - Strutture (`struct`)
 - Enumerativi (`enum`)

3.1.18.2 Tipi valore vs tipi riferimento

```
public struct Size
{
    public int height;
    public int width;
}
public class CSize
{
    public int height;
    public int width;
}
public static void Main()
{
    Size v;           // v istanza di Size
    v.height = 100;   // ok
    CSize r;          // r è un reference
    r.height = 100;   // NO, r non assegnato
    r = new CSize();  // r fa riferimento a un CSize
    r.height = 100;   // ok, r inizializzata
}
```

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA



Appena dichiaro la variabile `r`, questa viene salvato nello Stack, poi, quando viene inizializzata, viene creato un riferimento della classe `Csize` nello Heap.

3.1.18.3 Common Type System

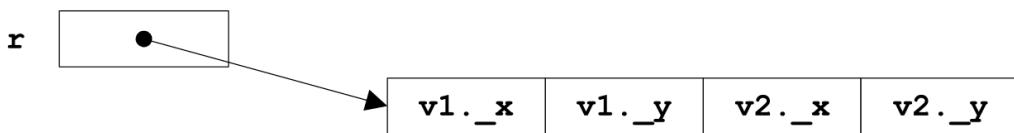
```
public struct Point
{
    private int _x, _y;
    public Point(int x, int y)
    {
        _x = x;
        _y = y;
    }
}
```

```

    }
    public int X
    {
        get { return _x; }
        set { _x = value; }
    }
    public int Y
    {
        get { return _y; }
        set { _y = value; }
    }
}

public class Rectangle
{
    Point v1;
    Point v2;
    ...
}
...
Rectangle r = new Rectangle();

```



```

    ...
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
    points[i] = new Point(i, i);
...

```

- Alla fine, rimane un solo oggetto nell'heap (l'array di Point)

```

    ...
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
{
    points[i].X = i;
    points[i].Y = i;
}
...

```

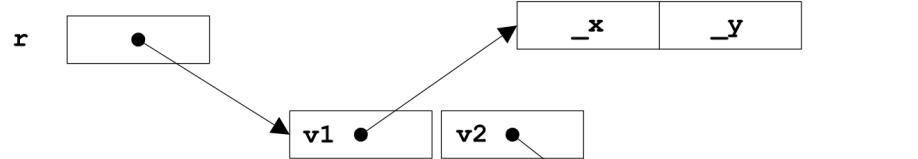
```

public class Point
{
    private int _x, _y;
    public Point(int x, int y)
    {
        _x = x;
        _y = y;
    }
    public int X
    {
        get { return _x; }
    }
}
```

```

        set { _x = value; }
    }
    public int Y
    {
        get { return _y; }
        set { _y = value; }
    }
}

```



```

public class Rectangle
{
    Point v1;
    Point v2;
    ...
}

...
Rectangle r = new Rectangle();

```

```

...
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
points[i] = new Point(i, i);
...

```

- Alla fine, rimangono 101 oggetti nell'heap (1 array di Point + 100 Point)

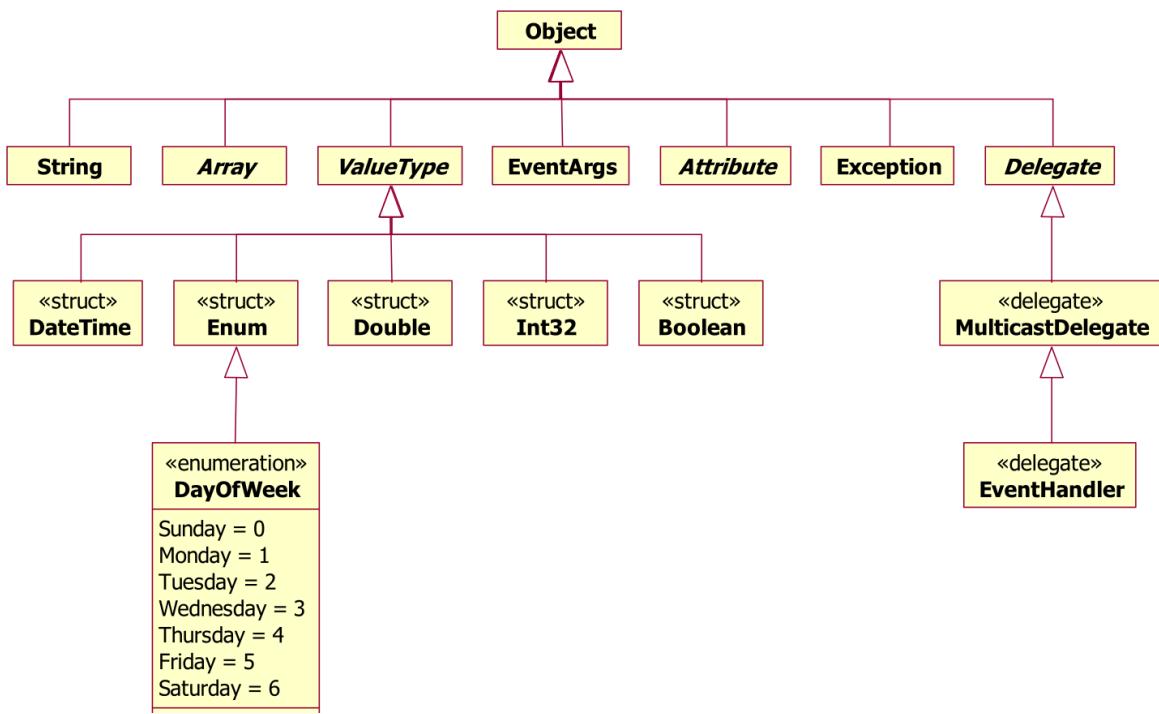
```

...
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
{
    points[i].X = i;
    points[i].Y = i;
}
...

```

NO!

3.1.18.4 Tipi valore e tipi riferimento



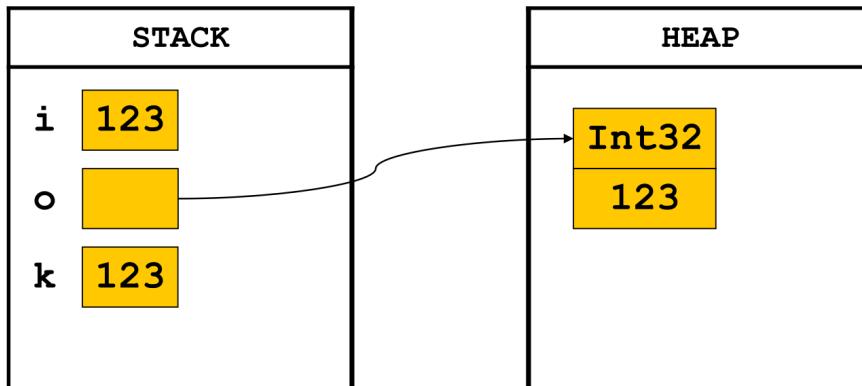
3.1.19 Boxing / Unboxing

- Un qualsiasi tipo valore può essere automaticamente convertito in un tipo riferimento (**boxing**) mediante un up cast implicito a `System.Object`

```
int i = 123;
object o = i;
```
- Un tipo valore “boxed” può tornare a essere un tipo valore standard (**unboxing**) mediante un down cast esplicito


```
int k = (int) o;
```
- Un tipo valore “boxed” è un **clone indipendente**, quindi se, dopo aver eseguito il boxing, cambio il valore di `i`, il valore di `k` non viene modificato

```
int i = 123;
object o = i;
int k = (int) o;
```



3.1.20 Bibliografia

Libri di base:

- D. S. Platt, *Introducing Microsoft® .NET*, Second Edition
- J. Sharp, J. Jagger, *Microsoft® Visual C#™ .NET Step by Step*
- T. Archer, A. Whitechapel, *Inside C#, Second Edition*

- M. J. Young, XML Stepby Step, Second Edition
- R. M. Riordan, Microsoft® ADO.NET Stepby Step

Libri avanzati:

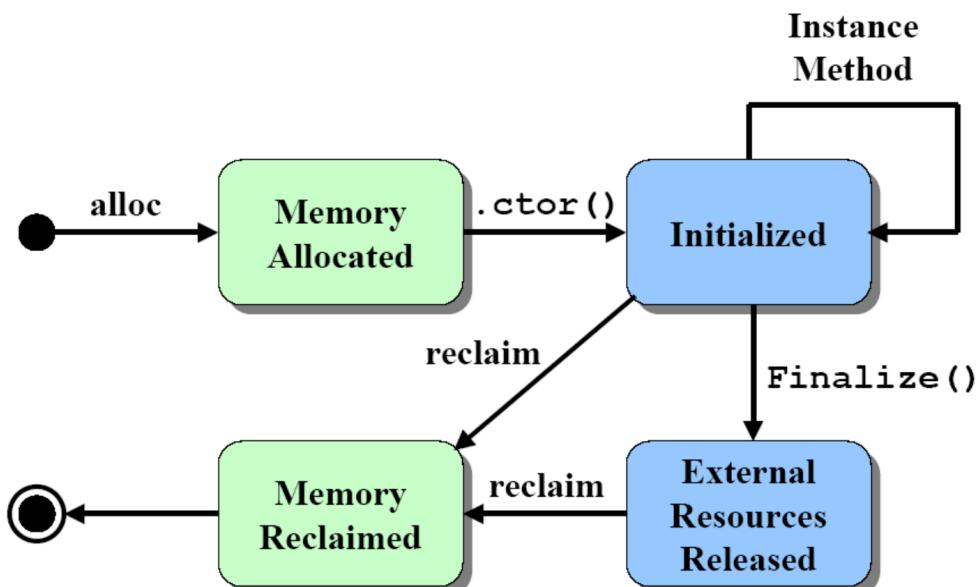
- J. Richter, AppliedMicrosoft® .NET Framework Programming
- C. Petzold, Programming Microsoft® Windows® with C#
- S. Lidin, Inside Microsoft® .NET IL Assembler

3.2 Garbage Collection

3.2.1 Utilizzo di un oggetto

- In un ambiente object-oriented, ogni oggetto che deve essere utilizzato dal programma
 - È descritto da un tipo
 - Ha bisogno di un'area di memoria dove memorizzare il suo stato
 - Una volta che si conosce il tipo dell'oggetto sappiamo quanta memoria allocare per lo stesso
- Passi per utilizzare un oggetto di tipo riferimento:
 - **Allocare memoria** per l'oggetto
 - **Inizializzare la memoria** per rendere utilizzabile l'oggetto
 - **Usare l'oggetto**
 - **Eseguire un clean up** dello stato dell'oggetto, se necessario; se, ad esempio, all'interno di quell'oggetto c'era un riferimento a un puntatore a file, quel file deve essere chiuso
 - **Liberare la memoria**

3.2.2 Ciclo di vita di un oggetto



3.2.3 Allocazione della memoria

- In C:
 - `malloc` (`calloc` , `realloc`)
- In C++:
 - `malloc` (`calloc` , `realloc`)
 - `new`
- In Java:

- ▶ new
- In IL:
 - ▶ newobj
- In C#:
 - ▶ new

3.2.4 Inizializzazione della memoria

- Definite Assignment: a ogni variabile deve essere sempre assegnato un valore prima che essa venga utilizzata
- il compilatore deve assicurarsi che ciò sia sempre verificato
- Data-flow analysis del codice
- valori di default
- usati in generale per tipi valore
- ad esempio, in Java le variabili di classe, locali e i componenti di un array sono inizializzati al valore di default, non le variabili di istanza, perché in quel caso è il costruttore che si deve occupare di inizializzarle
- costruttore
- usato per i tipi classe (Java, C++, C#)

3.2.5 Definite Assignment

```
int k;
if (v > 0 && (k = System.in.read()) >= 0)
    System.out.println(k);
```

Questo è corretto

```
int k;
while (n < 4) {
    k = n;
    if (k >= 5) break;
    n = 6;
}
System.out.println(k);
```

Questo è sbagliato, perché il compilatore non sa il valore di n

```
int k;
while (true) {
    k = n;
    if (k >= 5) break;
    n = 6;
}
System.out.println(k);
```

Questa è corretta

```
int k;
int n=5;
if (n>2) k=3;
System.out.println(k);
```

Questo non è corretto perché, a differenza dell'esempio sopra, l'espressione all'interno dell'if non è costante; se pensiamo a n come variabile condivisa, il valore di n potrebbe cambiare tra l'assegnamento e la valutazione.

3.2.6 Clean up dello stato

- In C++/C#:
 - distruttore (più propriamente, finalizzatore): ~{nome della classe}
 - unico, non ereditabile, no overload, senza parametri e modificatori
 - invocato automaticamente alla distruzione dell'oggetto (non può essere invocato)
- In java:
 - `finalize()`
 - metodo di `Object`
 - invocato automaticamente alla distruzione dell'oggetto (non può essere invocato)
 - il momento in cui viene invocato un finalizzatore dipende dalla JVM

3.2.7 Liberazione della memoria

- In C:
 - `free()`
- In C++:
 - `free()`
 - `delete`
- In java/C#: garbage collector (GC)

3.2.8 Cos'è il Garbage Collection

- Modalità automatica di rilascio delle risorse utilizzate da un oggetto
- Migliora la stabilità dei programmi
 - Evita errori connessi alla necessità, da parte del programmatore, di manipolare direttamente i puntatori alle aree di memoria
- Pro:
 - dangling pointer: puntatore che fa riferimento a un'area di memoria non più valida
 - doppia de-allocazione
 - memory leak
- Contro:
 - aumentata richiesta risorse di calcolo
 - incertezza del momento in cui viene effettuata la GC
 - rilascio della memoria non deterministico; quindi non posso distruggere un oggetto quando voglio io
- Strategie disponibili:
 - Tracing
 - determinare quali oggetti sono (potenzialmente) raggiungibili
 - eliminare gli oggetti non raggiungibili
 - Reference counting

- ogni oggetto contiene un contatore che indica il numero di riferimenti a esso
- la memoria può essere liberata quando il contatore raggiunge lo 0
- Escape analysis
 - si spostano oggetti dallo heap allo stack
 - l'analisi viene effettuata a compile-time in modo da stabilire se un oggetto, allocato all'interno di una subroutine, non è accessibile al di fuori di essa
 - riduce il lavoro del GC

3.2.9 GC: Reference counting

- Svantaggi:
 - Cicli di riferimenti
 - se due oggetti si referenziano a vicenda, il loro contatore non raggiungerà mai 0
 - Aumento dell'occupazione di memoria
 - Riduzione della velocità delle operazioni sui riferimenti
 - ogni operazione su un riferimento deve anche incrementare/decrementare i contatori
 - Atomicità dell'operazione
 - ogni modifica a un contatore deve essere resa operazione atomica in ambienti multi-threaded
 - Assenza di comportamento real-time
 - ogni operazione sui riferimenti può (potenzialmente) causare la de-allocazione di diversi oggetti
 - il numero di tali oggetti è limitato solamente dalla memoria allocata

3.2.10 GC: Tracing

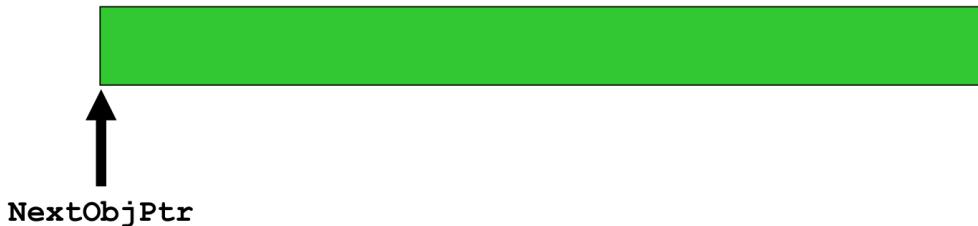
- Siano *p* e *q* due oggetti
- Sia *q* un oggetto raggiungibile
- Diremo che *p* è raggiungibile in maniera ricorsiva se e solo se:
 - esiste un riferimento a *p* tramite *q*
 - ovvero *p* è raggiungibile attraverso un oggetto, a sua volta raggiungibile
- Un oggetto è pertanto raggiungibile in due soli casi:
 - è un oggetto radice
 - creato all'avvio del programma (oggetto globale)
 - creato da una sub-routine (oggetto scope, riferito da variabile sullo stack)
 - è referenziato da un oggetto raggiungibile
 - la raggiungibilità è una chiusura transitiva
- La definizione di garbage tramite la raggiungibilità non è ottimale
 - può accadere che un programma utilizzi per l'ultima volta un certo oggetto molto prima che questo diventi irraggiungibile
- Distinzione:
 - garbage **sintattico**
(oggetti che il programma non può raggiungere)

- ▶ **garbage semantico**
 (oggetti che il programma non vuole più usare)
 - problema solo parzialmente decidibile, quindi non possono essere creati algoritmi per risolvere questo problema

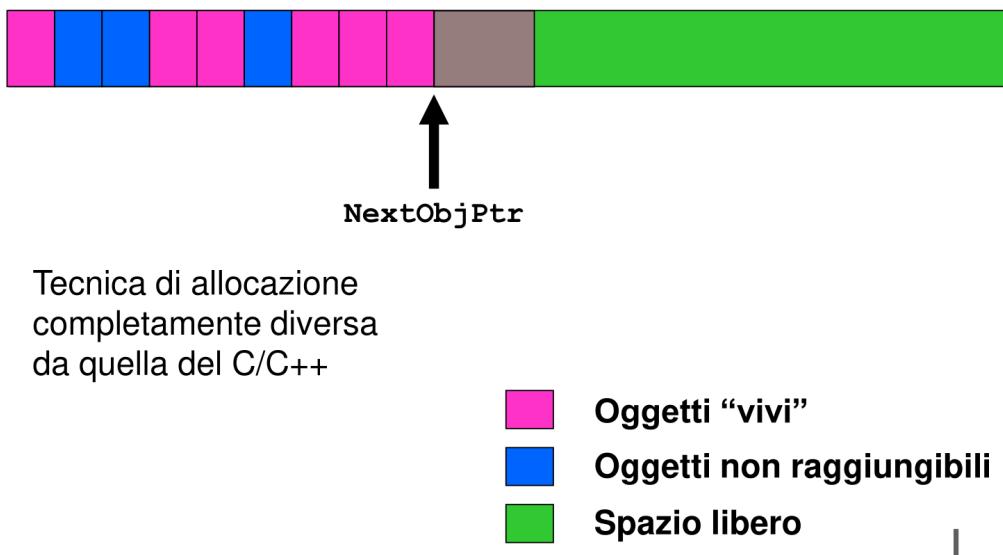
```
Object x = new Foo();
Object y = new Bar();
x = new Quux(); // qui l'oggetto Foo è garbage sintattico
if(x.check_something())
x.do_something(y); // qui y *potrebbe* essere garbage
semantico
```

3.2.11 Allocazione della memoria

- In fase di inizializzazione di un processo, il CLR
 - ▶ Riserva una regione contigua di spazio di indirizzamento managedheap
 - ▶ Memorizza in un puntatore (NextObjPtr) l'indirizzo di partenza della regione



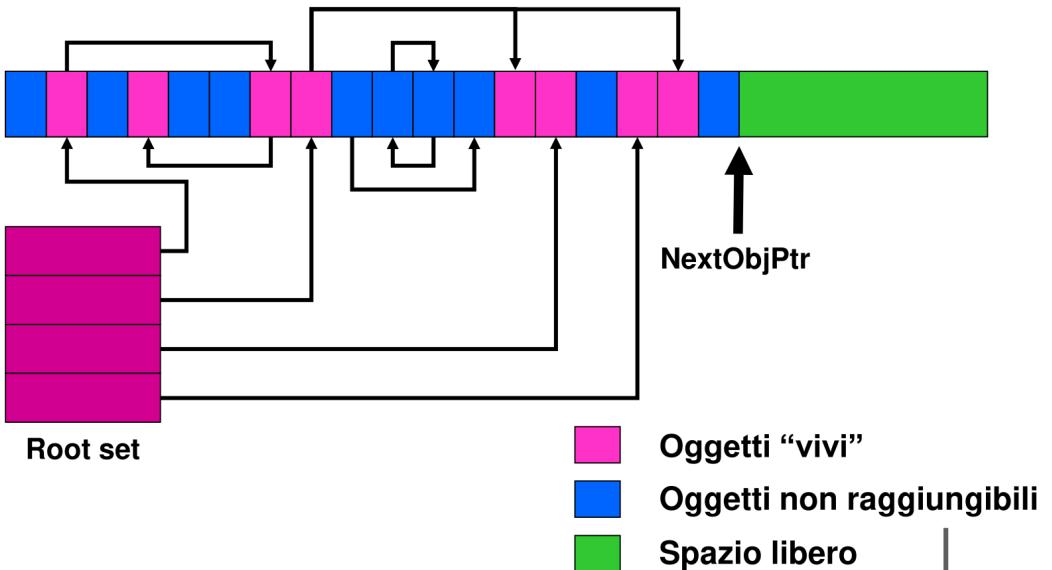
- Quando deve eseguire una newobj, il CLR
 - ▶ Calcola la dimensione in byte dell'oggetto e aggiunge all'oggetto due campi di 32 (o 64) bit
 - Un puntatore alla tabella dei metodi
 - Un campo SyncBlockIndex
 - ▶ Controlla che ci sia spazio sufficiente a partire da NextObjPtr
 - in caso di spazio insufficiente:
 - garbage collection
 - OutOfMemoryException
 - ▶ thisObjPtr= NextObjPtr;
 - ▶ NextObjPtr += sizeof(oggetto);
 - ▶ Invoca il costruttore dell'oggetto (this ≡ thisObjPtr)
 - ▶ Restituisce il riferimento all'oggetto



3.2.12 Garbage Collector

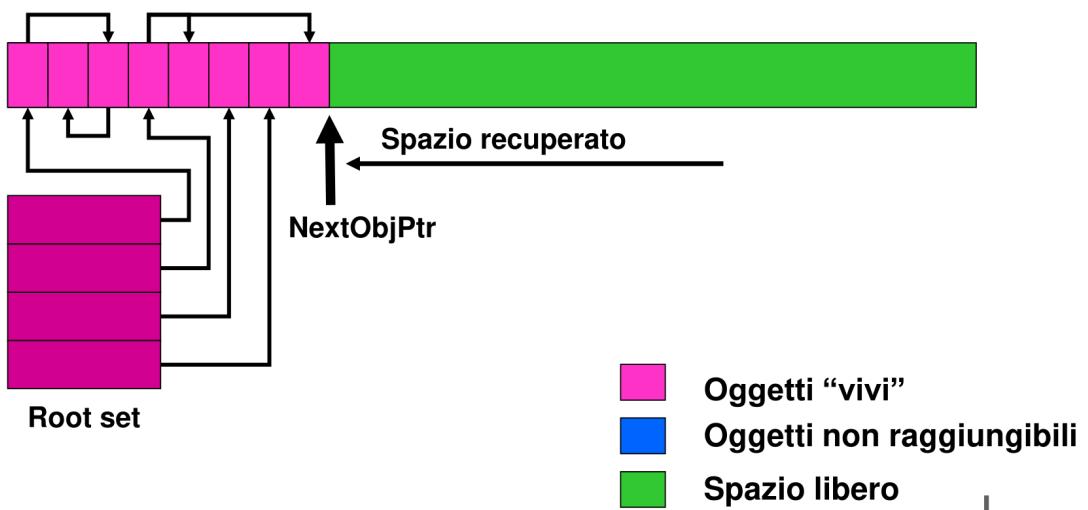
- Verifica se nell'heap esistono oggetti non più utilizzati dall'applicazione
- Ogni applicazione ha un insieme di radici (*root*)
- Ogni radice è un puntatore che contiene l'indirizzo di un oggetto di tipo riferimento oppure vale `null`
- Le radici sono:
 - Variabili globali e field statici di tipo riferimento
 - Variabili locali o argomenti attuali di tipo riferimento sugli stack dei vari thread
 - Registri della CPU che contengono l'indirizzo di un oggetto di tipo riferimento
- **Gli oggetti “vivi”** sono quelli **raggiungibili** direttamente o indirettamente dalle radici
- **Gli oggetti garbage** sono quelli **NON raggiungibili** direttamente o indirettamente dalle radici
- Quando parte, il GC ipotizza che tutti gli oggetti siano garbage
- Quindi, scandisce le radici e per ogni radice **marca**
 - l'eventuale oggetto referenziato e
 - Tutti gli oggetti a loro volta raggiungibili a partire da tale oggetto
- Se durante la scansione incontra un oggetto già marcato in precedenza, lo salta
 - sia per motivi di prestazioni
 - sia per gestire correttamente riferimenti ciclici
- Una volta terminata la scansione delle radici, tutti gli oggetti NON marcati sono non raggiungibili e quindi garbage

3.2.12.1 Fase 1: Mark



- Rilascia la memoria usata dagli oggetti non raggiungibili
- **Compatte** la memoria ancora in uso, **modificando nello stesso tempo tutti i riferimenti agli oggetti spostati!** Quindi sposta tutti gli oggetti in ordine dall'inizio dell'heap.
- Unifica la memoria disponibile, aggiornando il valore di `NextObjPtr`
- Tutte le operazioni che il GC effettua sono possibili in quanto
 - Il tipo di un oggetto è sempre noto
 - È possibile utilizzare i metadati per determinare quali field dell'oggetto fanno riferimento ad altri oggetti

3.2.12.2 Fase 2: Compact



3.2.13 Finalization

- Non è responsabilità del GC, ma del programmatore
- Se un oggetto contiene esclusivamente
 - tipi valore e/o
 - riferimenti a oggetti managed
- (maggior parte dei casi), non è necessario eseguire alcun codice particolare
- Se un oggetto contiene almeno un riferimento a un oggetto *unmanaged* (in genere, una risorsa del S.O.)

- file, connessione a database, socket, mutex, bitmap, ...

è necessario eseguire del codice per rilasciare la risorsa, prima della deallocazione dell'oggetto
- Ad esempio, un oggetto di tipo `System.IO.FileStream`
 - ▶ Prima deve aprire un file e memorizzare in un suo field l'handle del file (una risorsa di S.O. unmanaged)
 - ▶ Quindi usa tale handle nei metodi **Read** e **Write**
 - ▶ Infine, deve rilasciare l'handle nel metodo **Finalize**
- In C#
 - ▶ NON è possibile definire il metodo **Finalize**
 - ▶ È necessario definire un **distruttore** (sintassi C++)

```
public class OSHandle
{
    // Field contenente l'handle della risorsa unmanaged
    private readonly IntPtr handle;

    public IntPtr Handle
    { get { return _handle; } }

    public OSHandle(IntPtr handle)
    { _handle = handle; }

    ~OSHandle()
    { CloseHandle(_handle); }

    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static bool CloseHandle(IntPtr handle);
}
```

- Il compilatore C# trasforma il codice del distruttore

```
~OSHandle()
{ CloseHandle(_handle); }
```

nel seguente codice (ovviamente in IL):

```
protected override void Finalize()
{
    try
    { CloseHandle(_handle); }
    finally
    { base.Finalize(); }
}
```

- L'invocazione del metodo **Finalize** non avviene in modo deterministico
- Inoltre, non essendo un metodo pubblico, il metodo **Finalize** non può essere invocato direttamente
- Nel caso di utilizzo di risorse che devono essere rilasciate appena termina il loro uso, questa situazione è problematica
- Si pensi a file aperti o a connessioni a database che vengono chiusi solo quando il GC invoca il corrispondente metodo **Finalize**

- In questi casi, è di fondamentale importanza rilasciare (Dispose) o chiudere (Close) la risorsa in modo deterministico

3.2.14 Rilascio deterministico senza gestione delle eccezioni

```
...
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};
FileStream fs;
fs = new FileStream("Temp.dat", FileMode.Create);
fs.Write(bytesToWrite, 0, bytesToWrite.Length);
fs.Close();
...
```

3.2.15 Rilascio deterministico con gestione delle eccezioni

```
...
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};
FileStream fs = null;
try
{
    fs = new FileStream("Temp.dat", FileMode.Create);
    fs.Write(bytesToWrite, 0, bytesToWrite.Length);
}
finally
{
    if(fs != null) fs.Close();
}
...
...
```

3.2.16 Il pattern Dispose

Se un tipo T vuole offrire ai suoi utilizzatori un servizio di clean up esplicito, deve implementare l'interfaccia IDisposable

```
public interface IDisposable
{
    void Dispose();
}
```

- I clienti di T possono utilizzare l'istruzione using

```
using (T tx = ...)
{
    utilizzo di tx... #Invocazione automatica di tx.Dispose()
}
```

3.2.17 Rilascio deterministico con using

```
...
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};
using (FileStream fs = new FileStream("Temp.dat", FileMode.Create))
{
    fs.Write(bytesToWrite, 0, bytesToWrite.Length);
}
...
...
```

- Il tipo della variabile definita nella parte iniziale di using deve implementare l'interfaccia IDisposable
- All'uscita del blocco 'using', viene sempre invocato automaticamente il metodo Dispose

3.2.18 Il pattern Dispose (altro esempio di utilizzo)

```
public class CursorReplacer : IDisposable
{
    private readonly Cursor _previous;
    public CursorReplacer()
    {
        _previous = Cursor.Current;
        Cursor.Current = Cursors.WaitCursor;
    }
    public void Dispose()
    {
        Cursor.Current = _previous;
    }
}

List<DbTableWrapper> tableWrappers = new List<DbTableWrapper>();
// Recupero di tutte le tabelle selezionate
using (CursorReplacer cursorReplacer = new CursorReplacer())
{
    foreach (DbServerWrapper serverWrapper in SelectedDbServerWrappers)
        foreach (DbCatalogWrapper catalogWrapper in
serverWrapper.SelectedDbCatalogWrappers)
            foreach (DbTableWrapper tableWrapper in
catalogWrapper.SelectedDbTableWrappers)
                {
                    tableWrappers.Add(tableWrapper);
                }
}
```

3.3 Tipi in .NET

- Dal punto di vista del modo in cui le istanze vengono gestite in memoria (rappresentazione, tempo di vita, ...), i tipi possono essere distinti in:
 - *Reference type*
 - *Value type*
- Dal punto di vista sintattico (sintassi del linguaggio C#), i tipi possono essere distinti in:
 - Classi - `class`
 - Interfacce - `interface`
 - Strutture - `struct`
 - Enumerativi - `enum`
 - Delegati - `delegate`
 - Array - `[]`
- In .NET, si concretizzano sempre in una classe (anche nel caso di tipi built-in e di interfacce)
- In generale, un tipo può contenere la definizione di 0+:
 - Costanti - sempre implicitamente associate al tipo
 - Campi (field) - read-only o read-write, associati alle istanze o al tipo
 - Metodi - associati alle istanze o al tipo
 - Costruttori - di istanza o di tipo
 - Operatori - sempre associati al tipo
 - Operatori di conversione - sempre associati al tipo

- ▶ Proprietà - associate alle istanze o al tipo
- ▶ Indexer - sempre associati alle istanze
- ▶ Eventi - associati alle istanze o al tipo
- ▶ Tipi - annidati

3.3.1 Modificatori di visibilità

Modificatore	Tipi a livello base	Tipi annidati
private	Non applicabile	Visibile nel tipo contenitore (default)
protected	Non applicabile	Visibile nel tipo contenitore e nei suoi sottotipi
internal	Visibile nell' <i>assembly</i> contenitore (default)	Visibile nell' <i>assembly</i> contenitore
protected internal	Non applicabile	Visibile sia nel tipo contenitore e nei suoi sottotipi, sia nell' <i>assembly</i> contenitore
public	Visibilità completa	Visibilità completa

Modificatore	Dati	Operazioni - Eventi
private	default	Visibile nel tipo contenitore (default)
protected	Applicare esclusivamente a costanti	Visibile nel tipo contenitore e nei suoi sottotipi
internal	ed eventualmente a campi read-only (è comunque preferibile l'accesso mediante una proprietà)	Visibile nell' <i>assembly</i> contenitore
protected internal		Visibile sia nel tipo contenitore e nei suoi sottotipi, sia nell' <i>assembly</i> contenitore
public		Visibilità completa

- Non sono applicabili nei seguenti casi:
 - ▶ Costruttori di tipo (statici) sempre inaccessibili - invocati direttamente dal CLR
 - ▶ Distruttori (finalizer) sempre inaccessibili - invocati direttamente dal CLR
 - ▶ Membri di interfacce sempre pubblici
 - ▶ Membri di enum sempre pubblici
 - ▶ Implementazione esplicita di membri di interfacce visibilità particolare (pubblici/privati), non modificabile
 - ▶ Namespace sempre pubblici

3.3.2 Regole

- **Massimizzare l'incapsulamento minimizzando la visibilità**
- Information hiding a livello di assembly
 - Dichiarare `public` solo i tipi significativi dal punto di vista concettuale
- Information hiding a livello di classe
 - Dichiarare `public` solo metodi, proprietà ed eventi significativi dal punto di vista concettuale
 - Dichiarare `protected` solo le funzionalità che devono essere visibili nelle classi derivate, ma non esternamente ad esempio, costruttori particolari, metodi e proprietà virtuali non `public`
- Information hiding a livello di field
 - Field `private` e proprietà `public`
 - Field `private` e proprietà `protected`

3.3.3 Costanti

- Una **costante** è un simbolo che identifica un valore che non può cambiare
- Il **tipo** della costante può essere solo un tipo considerato primitivo dal CLR (compreso `string`)
- Il **valore** deve essere determinabile *a tempo di compilazione*
- Ad esempio, in `Int32` esistono:

```
public const int.MaxValue = 2147483647;
public const int.MinValue = -2147483648;
```

- In una classe contenitore di dimensioni prefissate, si potrebbe definire:

```
public const int MaxEntries = 100; // Warning!
```

- Si noti l'utilizzo della maiuscola iniziale
- È possibile applicare `const` anche alle variabili locali

3.3.4 Field

- Un **field** è un *data member* che può contenere:
 - un **valore** (un'istanza di un *value type*)
 - un **riferimento** (a un'istanza di un *reference type*) in genere, la realizzazione di un'associazione
- Può essere:
 - di **istanza** (*default*)
 - di **tipo** (*static*)
- Può essere:
 - **read-write** (*default*)
 - **read-only** (*readonly*)
 - inizializzato nella definizione o nel costruttore
- Esiste sempre un **valore di default** (`0`, `0.0`, `false`, `null`)
- Qual è la differenza tra le seguenti definizioni:

```
public const int MaxEntries = 100;
public static readonly int MaxEntries = 100;
```

- Nel primo caso, la costante `MaxEntries` viene “**iniettata**” nel codice del cliente

- se il valore viene modificato e se il cliente e il fornitore sono in assembly diversi, **è necessario ricompilare anche il codice del cliente**
- Nel secondo caso, l'accesso al `fieldMaxEntries` è quello standard: il valore è in memoria ed è necessario reperirlo
 - se il valore viene modificato e se il cliente e il fornitore sono in *assembly* diversi, **NON è necessario ricompilare anche il codice del cliente**

3.3.5 Regole

- Definire `const` solo le costanti “**vere**”, cioè i valori veramente immutabili nel tempo (nelle versioni del programma), negli altri casi utilizzare field statici `read-only`
 - il valore di `MaxEntries` non è una costante “vera” perché in una versione successiva del programma potrebbe cambiare
- **Costanti**
 - il nome dovrebbe iniziare con una lettera maiuscola
 - di solito, dovrebbe essere pubblica (ma non è sempre così)
- **Field**
 - il nome dovrebbe iniziare con “`_`” seguito da una lettera minuscola
 - deve essere privata (accesso sempre mediante proprietà)
- **Field read-only**
 - scegliere, a seconda delle situazioni, una delle due convenzioni precedenti

3.3.6 Modificatori di metodi

- `virtual`
 - `abstract`
 - `override`
 - `override sealed / sealed override`
 - Applicabili a:
 - **Metodi**
 - **Proprietà** (metodi `get` e `set`)
 - **Indexer** (metodi `get` e `set`)
 - **Eventi** (metodi `add` e `remove`)
- di istanza (cioè non statici)

3.3.7 Modificatore `virtual`

- **L'implementazione di un metodo virtuale può essere modificata** da un membro `override` di una classe derivata (discendente)
- Quando il metodo virtuale viene invocato, viene valutato il tipo run-time dell'oggetto su cui è invocato per vedere la presenza di un membro sovrascritto
 - Late binding
 - Polimorfismo
- **Per default, i metodi non sono virtuali**

```
protected virtual void Method()
{ ... }
public virtual int Property
{ get { ... } set { ... } }
public virtual int this[int index]
{ get { ... } }
```

3.3.8 Modificatore **abstract**

- Si usa il modificatore **abstract** per indicare che il metodo non contiene alcuna implementazione
- I metodi astratti hanno le seguenti caratteristiche
 - Un metodo virtuale è implicitamente virtuale
 - La dichiarazione di metodi astratti è permessa solo in classi astratte
- L'implementazione di un metodo astratto verrà fornita da un metodo sovrascrivente

```
protected abstract void Method();
public abstract int Property
{ get; set; }
public abstract int this[int index]
{ get; }
```

3.3.9 Modificatore **override**

- Un metodo **override** **fornisce una (nuova) implementazione** di un metodo ereditato da una classe base
 - Il metodo sovrascritto da una dichiarazione **override** è detto metodo **basesovrascritto(overridden)**
- Il metodo base sovrascritto
 - Deve essere **virtual**, **abstract**, o **override**
 - Deve avere la stessa firma (signature) del metodo **override**
- Una dichiarazione **override** **non può cambiare l'accessibilità** del metodo base sovrascritto (diverso da Java)
- L'uso del modificatore **sealed** impedisce a una qualsiasi classe derivata l'ulteriore sovrascrittura del metodo

```
protected override void Method()
{ ... }
public override sealed int Property
{ get { ... } set { ... } }
public override int this[int index]
{ get { ... } }
```

3.3.10 Metodi

3.3.10.1 Passaggio degli argomenti

- Tre tipi di argomenti:
 - **In** (default in C#)
 - L'argomento deve essere inizializzato
 - L'argomento viene passato per **valore** (per **copia**)
 - Eventuali modifiche del valore dell'argomento **non hanno effetto** sul chiamante
 - **In/Out** (ref in C#)
 - L'argomento deve essere inizializzato
 - L'argomento viene passato per **riferimento**
 - Eventuali modifiche del valore dell'argomento **hanno effetto** sul chiamante
 - **Out** (out in C#)
 - L'argomento può NON essere inizializzato

- L'argomento viene passato per **riferimento**
- Le modifiche del valore dell'argomento (l'inizializzazione è obbligatoria) **hanno effetto** sul chiamante

3.3.10.2 Passaggio degli argomenti In

- **Value type**
 - ▶ Viene passata una copia dell'oggetto
 - ▶ Eventuali modifiche vengono effettuate sulla copia e **non hanno alcun effetto** sull'oggetto originale
- **Reference type**
 - ▶ Viene passata una copia del riferimento all'oggetto
 - ▶ Eventuali modifiche dell'oggetto referenziato hanno effetto
 - ▶ Eventuali modifiche del riferimento vengono effettuate sulla copia e **non hanno alcun effetto** sul riferimento originale

```
Point p1 = new Point(0,0);
Method1(p1);
Console.WriteLine("{0}",p1);
```

```
static void Method1(Point p)
{
    p.X = 100; p.Y = 100;
}
```

- Se Point è una classe le modifiche vengono effettuate sull'oggetto referenziato (100,100)
- Se Point è una struttura le modifiche vengono effettuate sulla copia (0,0)

3.3.10.3 Passaggio degli argomenti In/Out

- **Value type**
 - ▶ Viene passato l'indirizzo dell'oggetto
 - ▶ Eventuali modifiche **agiscono direttamente sull'oggetto** originale
- **Reference type**
 - ▶ Viene passato l'indirizzo del riferimento all'oggetto
 - ▶ Eventuali modifiche dell'oggetto referenziato **hanno effetto**
 - ▶ Eventuali modifiche del riferimento **agiscono direttamente sul riferimento originale**

```
Point p1 = new Point(0,0);
Method2(ref p1);
Console.WriteLine("{0}",p1);
```

```
static void Method2(ref Point p)
{
    p.X = 100; p.Y = 100;
}
```

- Se Point è una classe (100,100)
- Se Point è una struttura (100,100)

3.3.10.4 Passaggio degli argomenti

```

class/struct Persona ...

...
Persona p1 = new Persona("Tizio"); // p1 == Tizio*

Method1(p1);

// p1 == Tizio

Method2(ref p1);

// p1 == Sempronio
...
static void Method1(Persona p)
{
    p = new Persona("Caio"); /* p == Caio
}

static void Method2(ref Persona p)
{
    p = new Persona("Sempronio"); /* p == Sempronio
}

```

- p1 all'inizio fa riferimento a un'istanza di Persona con nome "Tizio"
- quando viene invocato `Method1()` nello scope della funzione la variabile p fa riferimento a un'istanza di Persona di nome "Caio", ma quando si esce dalla funzione il record di attivazione viene distrutto e quindi nel `main` p1 fa ancora riferimento all'istanza di Persona di prima, di nome "Tizio"
- Nella funzione `Method2()` viene creato un riferimento tra p e p1, cioè la variabile p punta alla variabile p1, che a sua volta punta all'istanza di Persona "Tizio"; quindi quando si invoca il metodo `new Persona()`, siccome tutte le operazioni fatte su p vengono fatte su p1, è la variabile p1 che punta a una nuova istanza di Persona "Sempronio"; quindi anche quando si esce dalla funzione e il record di attivazione viene distrutto, la variabile p1 continua a puntare all'istanza di Persona "Sempronio"

3.3.10.5 Passaggio degli argomenti Out

- **Value type e Reference type**

- Viene passato l'indirizzo dell'oggetto o del riferimento all'oggetto come nel caso In/Out
- Non è necessario che l'oggetto o il riferimento siano inizializzati prima di essere passati come argomento
- L'oggetto o il riferimento **DEVONO essere inizializzati** nel metodo a cui sono stati passati come argomento

```

Point p1;
Method3(out p1);

static void Method3(out Point p)
{
    // In questo punto il compilatore suppone che
    // p NON sia inizializzato
    p.X = 100; p.Y = 100; // Errore di compilazione!

```

```
p = new Point(100,100); // È indispensabile  
}
```

3.3.10.6 Regole

- Utilizzare prevalentemente il passaggio *standard* per valore
- Utilizzare il passaggio per riferimento (*ref* o *out*) solo se strettamente necessario
 - 2+ valori da restituire al chiamante
 - 1+ valori da utilizzare e modificare nel metodo
 - Scegliere *ref* se l'oggetto passato come argomento deve essere già stato inizializzato
 - Scegliere *out* se è responsabilità del metodo inizializzare completamente l'oggetto passato come argomento

3.3.10.7 Esempi

```
public static void Swap<T>(ref T arg1, ref T arg2)  
{  
    T temp = arg1;  
    arg1 = arg2;  
    arg2 = temp;  
}  
  
public static void SplitCognomeNome(string cognomeNome,  
        out string cognome, out string nome)  
{  
    string[] words = cognomeNome.Split(' ');  
    if (words.Length == 2)  
    {  
        cognome = words[0];  
        nome = words[1];  
    }  
    else  
    {  
        ...  
    }  
}
```

3.3.10.8 Numero variabile di argomenti

- Si supponga di dover scrivere:

```
Add(a,b); // a+b  
Add(10,20,30); // 10+20+30  
Add(x1,x2,x3,x4); // x1+x2+x3+x4
```

- Soluzioni possibili:
 - Overloading del metodo **Add**
 - **Svantaggio:** posso solo codificare un numero finito di metodi
 - Definire un solo metodo Add che accetti un numero variabile di argomenti

```
int Add(params int[] operands)  
{  
    int total = 0;  
    foreach (int operand in operands)  
        total += operand;  
    return total;  
}
```

- Non solo posso scrivere:

```
Add(a,b);
Add(10,20,30);
Add(x1,x2,x3,x4);
```

- Ma anche:

```
Add(); // restituisce 0
int[] numbers = { 10,20,30,40,50 };
Add(numbers);
Add(new int[] { 10,20,30,40,50 });
Add(new int[] { x1,x2,x3,x4,x5 });
```

- Zucchero sintattico:

```
Add(x1,x2,x3,x4);
```

è uguale a

```
Add(new int[] { x1,x2,x3,x4 });
```

3.3.11 Costruttori di istanza

- **Responsabilità:** inizializzare correttamente lo stato dell'oggetto appena creato (nulla di più!)
- In mancanza di altri costruttori, esiste sempre un costruttore di *default* senza argomenti che, semplicemente, invoca il costruttore senza argomenti della classe base
- Nel caso delle **classi**, il costruttore senza argomenti può essere definito dall'utente
- Nel caso delle **strutture**, il costruttore senza argomenti NON può essere definito dall'utente (per motivi di efficienza)
- In entrambi i casi, è possibile definire altri costruttori

con differente signaturee differente visibilità

```
public abstract class DataAdapterManager
{
    private readonly DataTable _dataTable;
    private readonly IConnectionManager _connectionManager;

    protected DataAdapterManager(DataTable dataTable,
        IConnectionManager connectionManager)
    {
        if(dataTable == null)
            throw new ArgumentNullException("dataTable");
        if(connectionManager == null)
            throw new ArgumentNullException("connectionManager");
        _dataTable = dataTable;
        _connectionManager = connectionManager;
    }
    ...
}
```

```
public class XmlDataAdapterManager : DataAdapterManager
{
    private readonly Encoding _encoding;
```

```

public XmlDataAdapterManager(IDataTable dataTable,
    XmlConnectionManager xmlConnectionManager)
: this(dataTable, xmlConnectionManager, Encoding.Unicode)
{ }

public XmlDataAdapterManager(IDataTable dataTable,
    XmlConnectionManager xmlConnectionManager,
    Encoding encoding)
: base(dataTable, xmlConnectionManager)
{
    _encoding = encoding;
}
...

```

3.3.12 Costruttori di tipo

- **Responsabilità:** inizializzare correttamente lo stato comune a tutte le istanze della classe - *field* statici
- Dichiарато **static**
- Implicitamente **private**
- Sempre senza argomenti - no *overloading*
- Può accedere esclusivamente ai membri (*field*, metodi, ...) statici della classe
- Se esiste, viene invocato automaticamente dal CLR
 - ▶ Prima della creazione della prima istanza della classe
 - ▶ Prima dell'invocazione di un qualsiasi metodo statico della classe
- Non basare il proprio codice sull'ordine di invocazione di costruttori di tipo

```

class MyType
{
    static int _x = 5; #Viene definito implicitamente un costruttore di tipo
    ...
}

class MyType
{
    static int _x;
    static MyType() { _x = 5; } #Del tutto analogo al caso precedente
    ...
}

class MyType
{
    static int _x = 5;
    static MyType() { _x = 10; } #_x viene prima inizializzato a 5 e quindi a 10
    ...
}

```

3.3.13 Regole

- Definire un costruttore di tipo solo se strettamente necessario, cioè se i campi statici della classe
 - NON possono essere inizializzati in linea
 - Devono essere inizializzati solo se la classe viene effettivamente utilizzata

```
public class A
{
    private static XmlDocument _xmlDocument;

    static A()
    {
        _xmlDocument = new XmlDocument();
        _xmlDocument.Load( ... );
    }

    ...
}
```

3.3.14 Costruttori ed eccezioni

- Supponiamo che
 - un costruttore lancia un'eccezione e
 - l'eccezione non venga gestita all'interno del costruttore stesso (quindi arriverà al chiamante)
- **Nel caso di costruttori di istanza** nessun problema!
In C++ è una situazione non facilmente gestibile
- **Nel caso di costruttori di tipo** la classe NON è più utilizzabile!
`TypeInitializationException`

3.3.15 Interfacce

- In C#, un'interfaccia può contenere esclusivamente:
 - **Metodi** - considerati pubblici e astratti
 - **Proprietà** - considerate pubbliche e astratte
 - **Indexer** - considerati pubblici e astratti
 - **Eventi** - considerati pubblici e astratti
- In CLR, un'interfaccia è considerata una particolare classe astratta di sistema che (ovviamente) non deriva da `System.Object`
 - però, le classi che la implementano derivano per forza da `System.Object`
- Un'interfaccia
 - Può essere implementata sia dai *reference type*, sia dai *value type*
 - È considerata sempre un **reference type**
 - **Attenzione:** se si effettua il cast di un *value type* a un'interfaccia, avviene un **boxing del value type** (con conseguente copia del valore)!

3.3.15.1 Implementazione di un'interfaccia

```
public interface IBehavior
{
    void Method();
    int Property { get; set; }
    int this[int index] { get; }
}

public class A : IBehavior
```

```

{
    public void Method() // virtual sealed
    { ... }
    public int Property // virtual sealed
    {
        get { ... }
        set { ... }
    }
    public int this[int index] // virtual sealed
    {
        get { ... }
    }
}

public class A : IBehavior
{
    public virtual void Method()
    { ... }
    public virtual int Property
    {
        get { ... }
        set { ... }
    }
    public virtual int this[int index]
    {
        get { ... }
    }
}

public class B : A
{
    public override void Method() ...
    public override int Property ...
    public override int this[int index] ...
}

```

3.3.15.2 Implementazione di un'interfaccia: classe astratta

```

public abstract class A : IBehavior
{
    public abstract void Method();
    public abstract int Property { get; set; }
    public abstract int this[int index] { get; }
}

public class B : A
{
    public override void Method() ...
    public override int Property ...
    public override int this[int index] ...
}

```

- Siccome A è una classe astratta, la sottoclasse B che la estende deve ridefinire tutti i metodi astratti di A

3.3.15.3 Implementazione esplicita di un'interfaccia

```

public class A : IBehavior
{

```

```

void IBehavior.Method()
{
    ...
int IBehavior.Property
{
    get { ... }
    set { ... }
}
int IBehavior.this[int index]
{
    get { ... }
}
}

```

```

A a = new A(...);
a.Method(); // Non compil!
((IBehavior) a).Method(); // Ok!

```

- Questo per
 - Name hiding
 - Avoiding name ambiguity

```

public interface IMyInterface1
{ void Close(); }

public interface IMyInterface2
{ void Close(); }

public class MyClass : IMyInterface1, IMyInterface2
{
    void IMyInterface1.Close()
    {...}
    void IMyInterface2.Close()
    {...}
    public void Close()
    {...}
}

MyClass a = new MyClass (...);
((IMyInterface1) a).Close(); // Ok!
((IMyInterface2) a).Close(); // Ok!
a.Close(); // Ok!

```

3.3.16 Interfaccia vs. Classe astratta

Interfaccia	Classe astratta
<p>Deve descrivere una funzionalità semplice, implementabile da oggetti eterogenei (cioè appartenenti a classi non correlate tra di loro)</p> <p>Ad esempio:</p> <ul style="list-style-type: none"> le istanze di tutte le classi che implementano l'interfaccia ICloneable sono clonabili le istanze di tutte le classi che implementano l'interfaccia IList sono trattate come collezioni 	<p>Può descrivere una funzionalità anche complessa, comune a un insieme di oggetti omogenei (cioè appartenenti a classi strettamente correlate tra di loro)</p> <p>Ad esempio:</p> <ul style="list-style-type: none"> la classe astratta Enum fornisce le funzionalità di base di tutti i tipi enumerativi

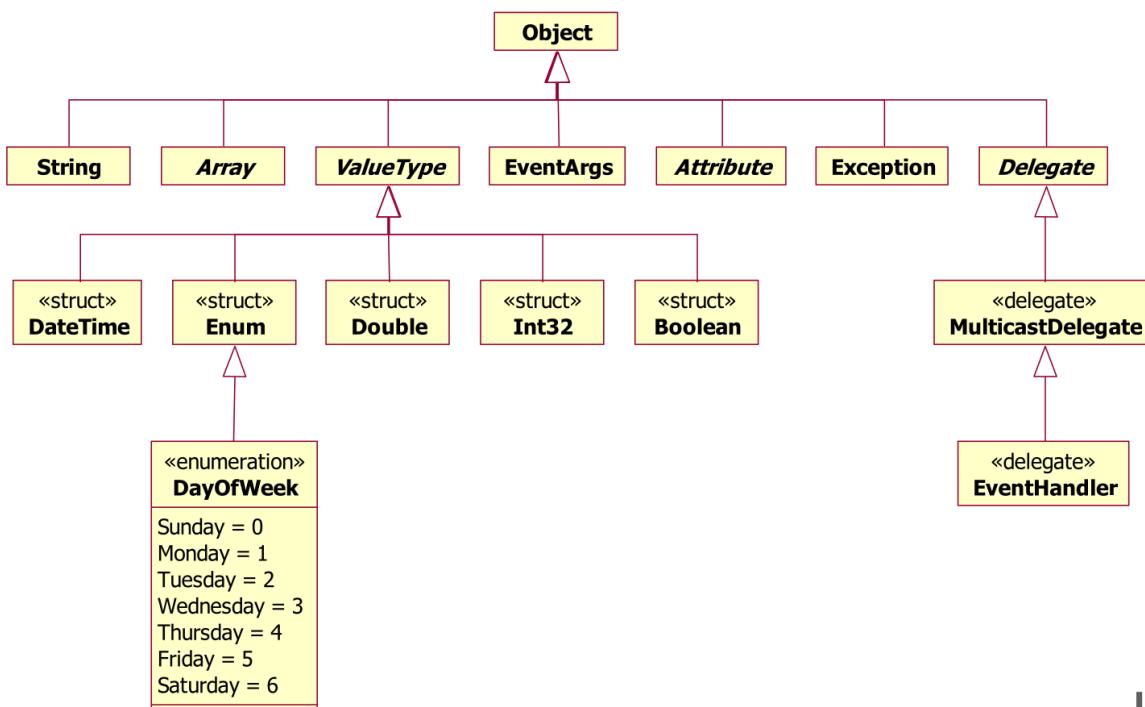
Interfaccia	Classe astratta
<p>Può “ereditare”</p> <ul style="list-style-type: none"> da 0+ interfacce 	<p>Può “ereditare”</p> <ul style="list-style-type: none"> da 0+ interfacce da 0+ classi (astratte e/o concrete) minimo 1 classe, se esiste una classe radice di sistema massimo 1 classe, se non è ammessa l'ereditarietà multipla
Non può essere istanziata	Non può essere istanziata
Non può contenere uno stato	Può contenere uno stato (comune a tutte le sottoclassi)
Non può contenere attributi membro e metodi (e proprietà ed eventi) statici (a parte eventuali costanti comuni)	Può contenere attributi membro e metodi (e proprietà ed eventi) statici

Interfaccia	Classe astratta
Non contiene alcuna implementazione	Può essere implementata completamente, parzialmente o per niente
<p>Le classi concrete che la implementano:</p> <ul style="list-style-type: none"> devono realizzare tutte le funzionalità 	<p>Le classi concrete che la estendono:</p> <ul style="list-style-type: none"> devono realizzare tutte le funzionalità non implementate possono fornire una realizzazione alternativa a quelle implementate
<p>Deve essere stabile</p> <p>Se si aggiungesse un metodo a un'interfaccia già in uso, tutte le classi che implementano quell'interfaccia dovrebbero essere modificate</p>	<p>Può essere modificata</p> <p>Quando si aggiunge un metodo a una classe astratta già in uso, è possibile fornire un'implementazione di default, in modo tale da non dover modificare le sottoclassi</p>

Interfaccia	Classe astratta
Non può gestire la creazione delle istanze delle classi che la implementano	Può gestire la creazione delle istanze delle sue sottoclassi
La creazione deve essere effettuata <ul style="list-style-type: none"> • dai costruttori delle suddette classi • da (un'istanza di) una classe non correlata, la cui unica funzionalità è la creazione di istanze di altre classi (classe factory) 	La creazione può essere effettuata <ul style="list-style-type: none"> • come per l'interfaccia, ma anche • da un metodo statico della classe astratta (metodo factory)

3.4 Classi e Interfacce Base

3.4.1 Framework .NET: Overview



3.4.2 System.Object

- La radice della gerarchia dei tipi
 - ▶ tutte le classi nel framework.NET Framework sono derivate da **Object**
- Ogni metodo definito nella classe **Object** è disponibile in tutti gli oggetti del sistema

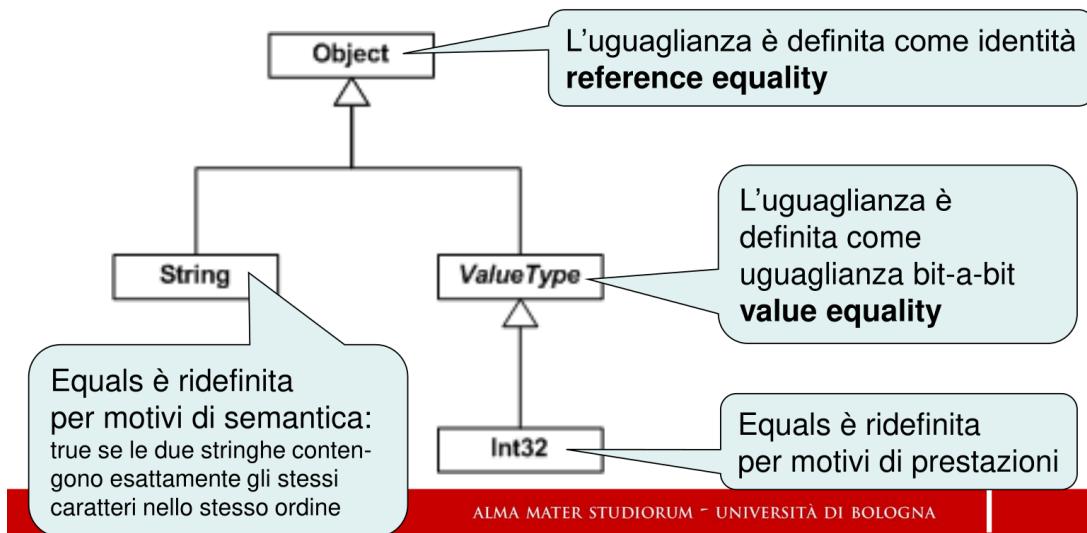
Object
+ Object () # Finalize () + GetHashCode () : System.Int32 + Equals ([in] obj : System.Object) : System.Boolean + ToString () : System.String + Equals ([in] objA : System.Object , [in] objB : System.Object) : System.Boolean + ReferenceEquals ([in] objA : System.Object , [in] objB : System.Object) : System.Boolean + GetType () : System.Type # MemberwiseClone () : System.Object

- Le classi derivate possono (e/o devono) sovrascrivere (override) alcuni di tali metodi, tra cui:

- Equals - Supporta il confronto tra oggetti
- ToString - Crea una stringa “comprendibile” che descrive un’istanza della classe
- GetHashCode - Genera un numero corrispondente al valore (stato) dell’oggetto per consentire l’uso di tabelle hash
- Finalize - Effettua operazioni di “pulizia” prima che un oggetto venga automaticamente distrutto

3.4.3 Object.Equals

- public virtual bool Equals(object obj);
- Valore di ritorno: true se this è uguale a obj, cioè se referenziano lo stesso oggetto, altrimenti false



- Le seguenti condizioni devono essere true per tutte le implementazioni del metodo `Equals`
 - Nell’elenco, x, y, e z rappresentano riferimenti non nulli a oggetti
 - `x.Equals(x)` restituisce true
 - `x.Equals(y)` restituisce lo stesso valore di `y.Equals(x)`
 - `x.Equals(y)` restituisce true se sia x che y sono NaN
 - `(x.Equals(y) && y.Equals(z))` restituisce true se e solo se `x.Equals(z)` restituisce true
 - Chiamate successive a `x.Equals(y)` restituiscono sempre lo stesso valore fintantoché gli oggetti riferenziati da x e y non vengono modificati
 - `x.Equals(null)` restituisce false
- Le implementazioni di `Equals` non devono generare eccezioni
- I tipi che sovrascrivono `Equals` devono anche sovrascrivere `GetHashCode`; altrimenti, `Hashtable` potrebbe non funzionare correttamente
- Se il linguaggio permette l’overloading di operatori e, per un certo tipo, si effettua l’overloading dell’operatore di uguaglianza, tale tipo deve anche sovrascrivere il metodo `Equals`. Tale implementazione del metodo `Equals` **deve restituire gli stessi risultati dell’operatore di uguaglianza**

```
public class Point
{
    private readonly int _x, _y;
    ...
    public override bool Equals(object obj)
```

```

{
//Check for null and compare run-time types.
if(obj == null || GetType() != obj.GetType())
    return false;
Point p = (Point) obj;
return (_x == p._x) && (_y == p._y);
}
public override int GetHashCode()
{
    return _x ^ _y;
}
}

```

3.4.4 Object.Equals

```

public class SpecialPoint : Point
{
    private readonly int _w;
    ...
    public SpecialPoint(int x, int y, int w) : base(x, y)
    {
        _w = w;
    }
    public override bool Equals(object obj)
    {
        return base.Equals(obj) &&
            _w == ((SpecialPoint) obj)._w;
    }
    public override int GetHashCode()
    {
        return base.GetHashCode() ^ _w;
    }
}

public class Rectangle
{
    private readonly Point _a, _b;
    ...
    public override bool Equals(object obj)
    {
        if(obj == null || GetType() != obj.GetType())
            return false;
        Rectangle r = (Rectangle) obj;
        // Uses Equals to compare variables.
        return _a.Equals(r._a) && _b.Equals(r._b);
    }
    public override int GetHashCode()
    {
        return _a.GetHashCode() ^ _b.GetHashCode();
    }
}

public struct Complex
{
    private readonly double _re, _im;
    ...
    public override bool Equals(object obj)
    {

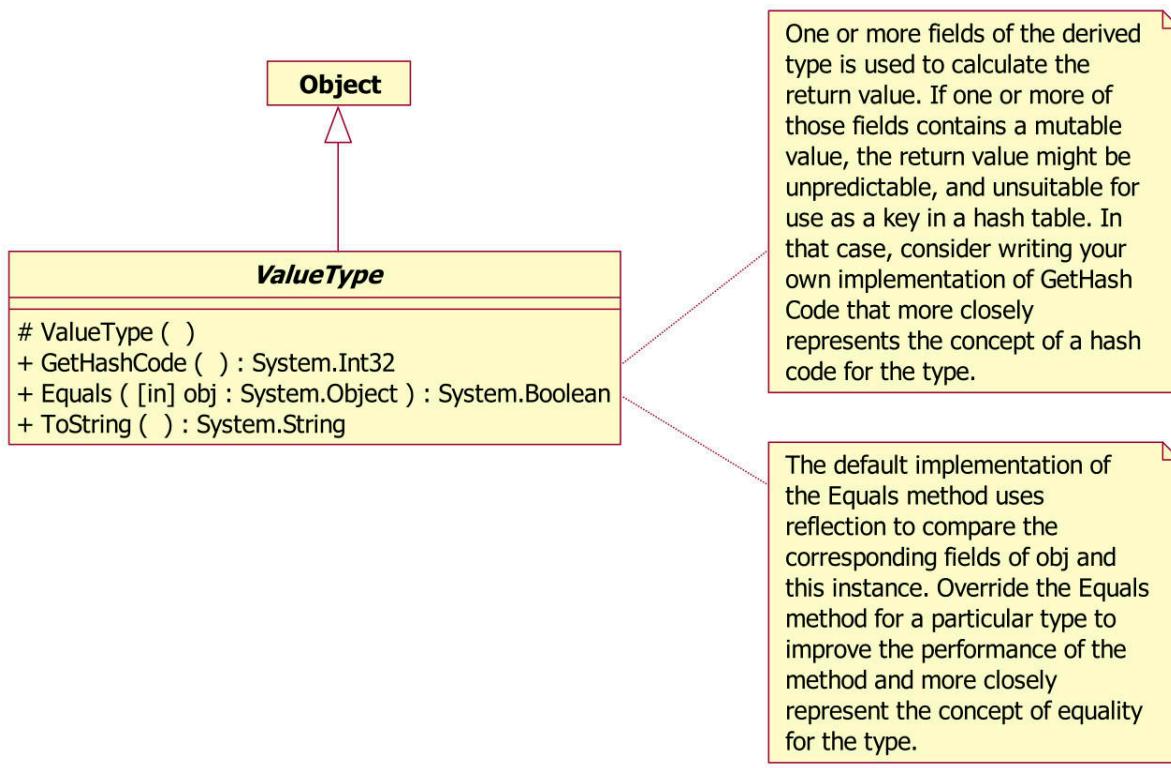
```

```

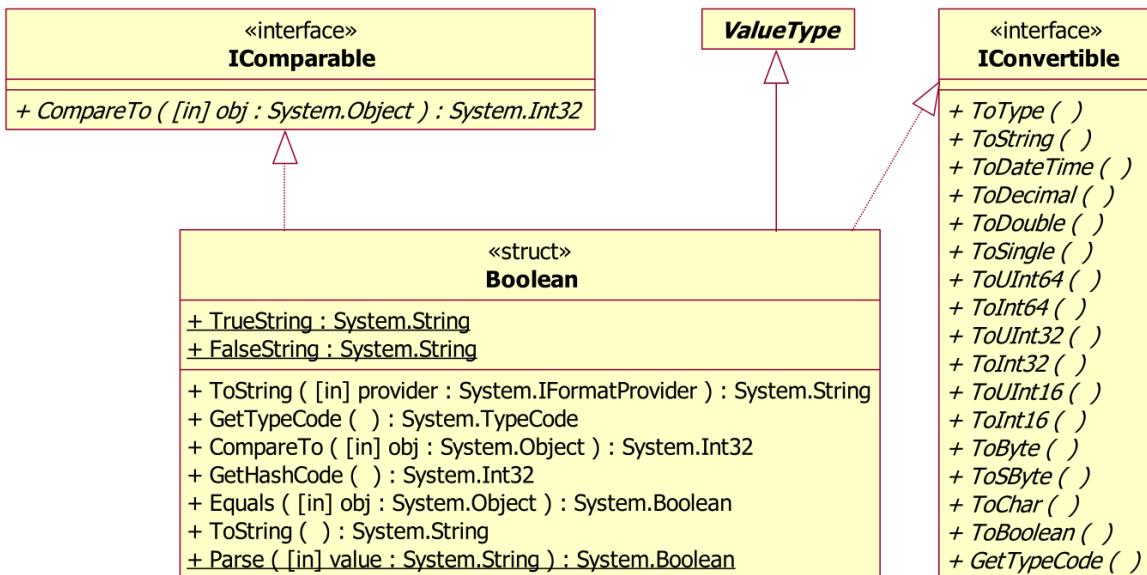
        return obj is Complex && this == (Complex) obj;
    }
    public override int GetHashCode()
    {
        return _re.GetHashCode() ^ _im.GetHashCode();
    }
    public static bool operator ==(Complex x, Complex y)
    {
        return x._re == y._re && x._im == y._im;
    }
    public static bool operator !=(Complex x, Complex y)
    {
        return !(x == y);
    }
}

```

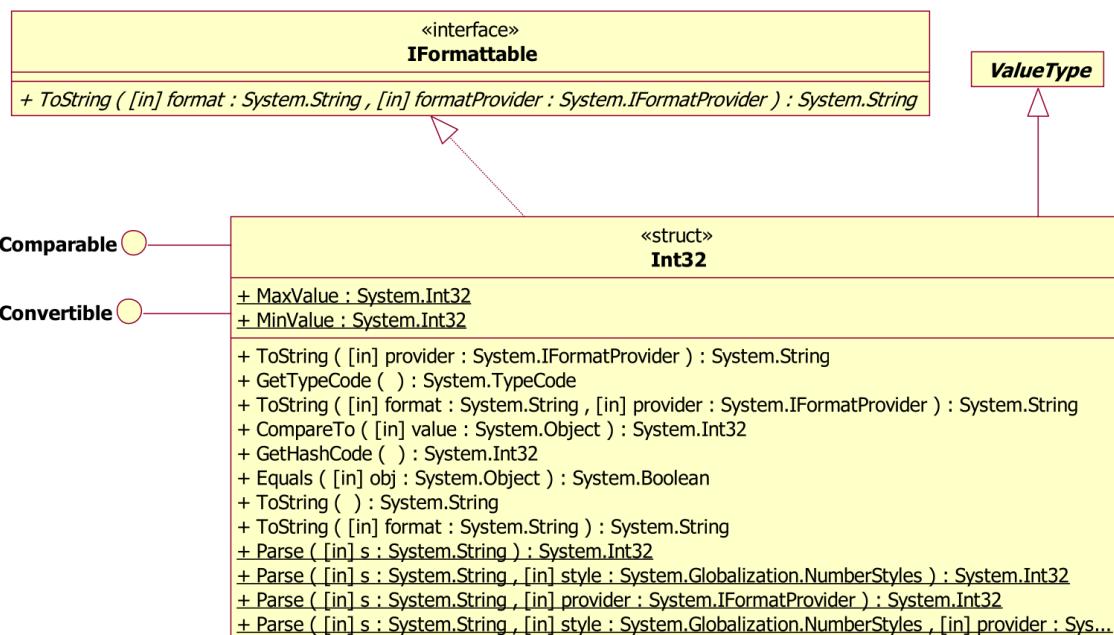
3.4.5 System.ValueType



3.4.6 System.Boolean

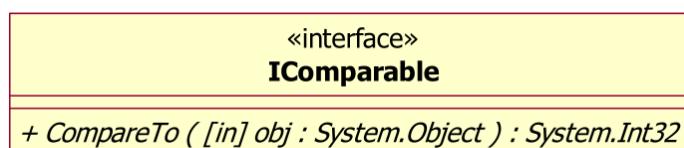


3.4.7 System.Int32



3.4.8 System.IComparable

- Confronta l'istanza corrente con un altro oggetto dello stesso tipo
- Valore di ritorno:** un 32-bit signed integer che indica l'ordine relativo degli oggetti confrontati
- La semantica del valore restituito è la seguente:
 - Minore di zero - l'istanza **this** precede **obj**
 - Zero - l'istanza **this** è uguale a **obj**
 - Maggiore di zero - l'istanza **this** segue **obj**
- Per definizione, ogni oggetto segue un riferimento **null**
- Il parametro **obj** deve essere dello stesso tipo della classe o valuetype che implementa questa interfaccia; altrimenti, va lanciata una **ArgumentException**



3.4.9 System.IComparable

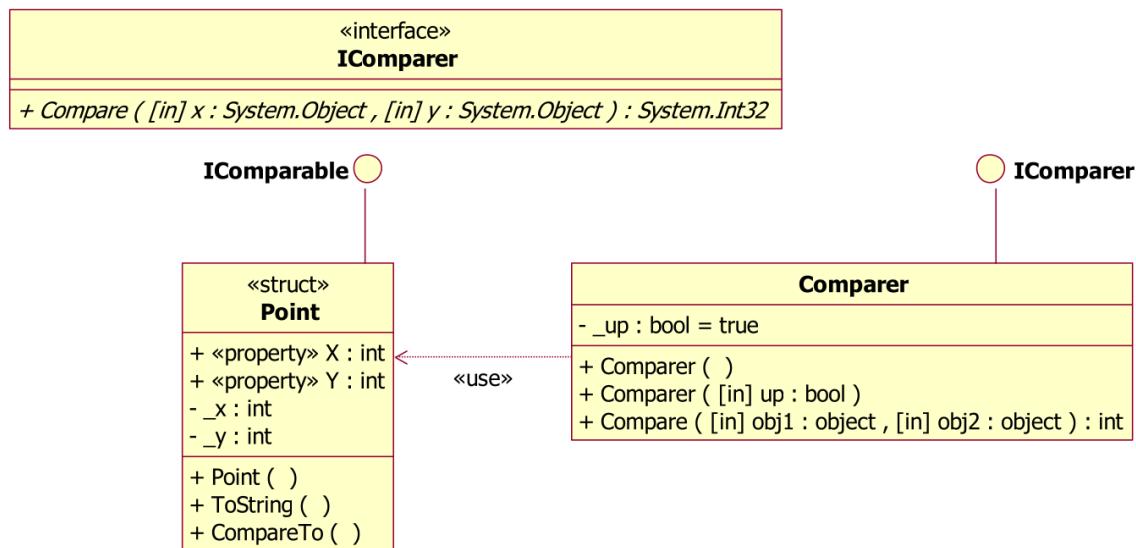
- Note per gli Implementatori:

Per ogni oggetto A, B e C, devono valere le seguenti condizioni:

- A.CompareTo(A) deve restituire zero
- Se A.CompareTo(B) restituisce zero allora anche B.CompareTo(A) deve restituire zero
- Se A.CompareTo(B) restituisce zero e B.CompareTo(C) restituisce zero allora anche A.CompareTo(C) deve restituire zero
- Se A.CompareTo(B) restituisce un valore diverso da zero allora B.CompareTo(A) deve restituire un valore dal segno opposto
- Se A.CompareTo(B) restituisce un valore x diverso da zero, e B.CompareTo(C) un valore y dello stesso segno di x, allora A.CompareTo(C) deve restituire un valore dello stesso segno di x e y
- Se volessi:
 - Ordinare i punti in ordine decrescente
 - Ordinare dei film
 - Per genere, oppure
 - Per titolo
 - Ordinare degli studenti
 - Per cognome e nome, oppure
 - Per matricola, oppure
 - Per corso di studio
 - ...

3.4.9.1 System.Collections.IComparer

- Questa interfaccia è usata, ad esempio, dai metodi `Array.Sort` e `Array.BinarySearch`
- Fornisce un modo per personalizzare il criterio di ordinamento



3.4.10 System.IConvertible

- Questa interfaccia fornisce metodi per convertire il valore di un'istanza di un tipo che implementa l'interfaccia in un valore equivalente di un tipo CLR

- I tipi **CLR** sono Boolean, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime, Char, e String
- Se non esiste una conversione sensata verso un tipo CLR, l’implementazione particolare del corrispondente metodo dell’interfaccia deve lanciare una InvalidCastException
- Ad esempio, questa interfaccia è implementata dal tipo Boolean; l’implementazione del metodo ToDateTimelancia un’eccezione in quanto non esiste alcun valore DateTimeequivalente a un valore del un tipo Boolean

«interface»
IConvertible
+ <i>ToType()</i>
+ <i>ToString()</i>
+ <i>ToDateTime()</i>
+ <i>ToDecimal()</i>
+ <i>ToDouble()</i>
+ <i>ToSingle()</i>
+ <i>ToUInt64()</i>
+ <i>ToInt64()</i>
+ <i>ToUInt32()</i>
+ <i>ToInt32()</i>
+ <i>ToUInt16()</i>
+ <i>ToInt16()</i>
+ <i>ToByte()</i>
+ <i>ToSByte()</i>
+ <i>ToChar()</i>
+ <i>ToBoolean()</i>
+ <i>GetTypeCode()</i>

3.4.11 System.Convert

- In System.Int32, l’implementazione dell’interfaccia System.IConvertible è un esempio di “*explicit interface implementation*”:

```
int x = 32;
double d = x.ToDouble(...); // No!
```

È necessario scrivere:

```
((IConvertible)x).ToDouble(...)
```

- Se necessario, utilizzare la classe Convert:

```
Convert.ToDouble(x)
```

Convert
+ DBNull : System.Object
+ GetTypeCode()
+ IsDBNull()
+ ChangeType()
+ ToBoolean()
+ ToChar()
+ ToSingle()
+ ToDouble()
+ ToDecimal()
+ ToDateTime()
+ ToByte()
+ ToSByte()
+ToInt16()
+ToUInt16()
+ToInt32()
+ToUInt32()
+ToInt64()
+ToUInt64()
+ToString()
+ToBase64String()
+ToBase64String()
+FromBase64String()
+ToBase64CharArray()
+FromBase64CharArray()

3.4.12 System.Convert

- Lancia un'eccezione se la conversione non è supportata

```
bool b = Convert.ToBoolean(DateTime.Today);
// InvalidCastException
```

- Effettua conversioni controllate

```
int k = 300;
byte b = (byte) k; // b == 44
byte b = Convert.ToByte(k); // OverflowException
▶ In alcuni casi, esegue un arrotondamento:
```

```
double d = 42.72;
int k = (int) d; // k == 42
int k = Convert.ToInt32(d); // k == 43
```

- È utile anche quando si ha una stringhe deve essere convertita in valore numerico:

```
string myString = "123456789";
int myInt = Convert.ToInt32(myString);
```

3.4.13 Conversione di tipo

- Una conversione di ampliamento** avviene quando un valore di un tipo viene convertito verso un altro tipo che è di dimensione uguale o superiore
 - Da Int32 a Int64
 - Da Int32 a UInt64
 - Da Int32 a Single (con possibile perdita di precisione)
 - Da Int32 a Double
- Una conversione di restrizione avviene quando un valore di un tipo viene convertito verso un altro tipo che è di dimensione inferiore
 - Da Int32 a Byte
 - Da Int32 a SByte
 - Da Int32 a Int16
 - Da Int32 a UInt16
 - Da Int32 a UInt32
- Conversioni implicite** - non generano eccezioni

► Conversioni numeriche

Il tipo di destinazione dovrebbe essere in grado di contenere, senza perdita di informazione, tutti i valori ammessi dal tipo di partenza

Eccezione:

```
int k1 = 1234567891;
float b = k1;
int k2 = (int) b; // k2 == 1234567936
```

► Up cast

Principio di sostituibilità: deve sempre essere possibile utilizzare una classe derivata al posto della classe base

```
B b = new B(...); // class B : A
A a = b;
```

- **Conversioni esplicite** - possono generare eccezioni

- **Conversioni numeriche** Il tipo di destinazione non sempre è in grado di contenere il valore del tipo di partenza

```
int k1 = -1234567891;
uint k2 = (uint) k1; // k2 == 3060399405

int k1 = -1234567891;
uint k2 = checked((uint) k1); // OverflowException

int k1 = -1234567891;
uint k2 = Convert.ToInt32(k1); // OverflowException
```

- **Conversioni esplicite** - possono generare eccezioni

- **Down cast**

```
A a = new B(...); // class B : A
B b = (B) a; // Ok

a = new A(...);
b = (B) a; // InvalidCastException

if(a is B) // if(a.GetType() == typeof(B))
{
    b = (B) a; // Non genera eccezioni
    ...
}

b = a as B; // b = (a is B)? (B) a : null;
if(b != null)
{
    ...
}
```

- **Boxing - up cast** (conversione implicita)

```
int k1 = 100;
object o = k1; // Copia!
k1 = 200;
```

- **Unboxing - down cast** (conversione esplicita)

```
int k2 = (int) o; // k1 = 200, k2 = 100
double d1 = (double) k1; // Ok
d1 = k1; // Ok
```

```
d1 = o; // Non compila!
d1 = (double) o; // InvalidCastException
d1 = (int) o; // Ok
```

3.4.13.1 Conversione di tipo definite dall'utente

```
public static implicit operator typeOut(typeIn obj)
public static explicit operator typeOut(typeIn obj)
```

- Metodi statici di una classe o di una struttura
- La keyword `implicit` indica l'utilizzo automatico (cast隐式的)

Il metodo non deve generare eccezioni

- La keyword `explicit` indica la necessità di un cast esplicito

Il metodo può generare eccezioni

- `typeOut` è il tipo del risultato del cast
- `typeIn` è il tipo del valore da convertire
- `typeIn` o `typeOut` deve essere il tipo che contiene il metodo

3.4.14 Conversioni a string

- Conversioni a string (di un Int32):

► `ToString()`

```
int k1 = -1234567891;
string str = k1.ToString(); // str == " - 1234567891"
```

► `ToString(string formatString)`

l'istanza è formattata secondo il `NumberFormatInfo` dell'impostazione cultura (*culture*) corrente

```
k1.ToString("X"); // = "B669FD2D"
k1.ToString("C"); // = " - € 1.234.567.891,00"
k1.ToString("C0"); // = " - € 1.234.567.891"
k1.ToString("N0"); // = " - 1.234.567.891"
k1.ToString("E"); // = " - 1,234568E+009"
```

- Conversioni a string (di un Int32):

► `String.Format(string format, params object[] args)`

Il parametro `format` è costituito da uno o più elementi di formato nella forma:

```
{index[,alignment][:formatString]}
```

```
int k1 = -1234567891;
String.Format ("{}{0}",k1); // = " - 1234567891"
String.Format ("{:0:X}{0}",k1); // = "B669FD2D"
String.Format ("{:0,5:X}{0}",k1); // = "B669FD2D"
String.Format ("{:0,10:X}{0}",k1); // = " ΔΔ B669FD2D"
String.Format ("{:0, - 10:X}{0}",k1); // = "B669FD2D ΔΔ "
String.Format ("{:0:N0}{0}",k1); // = " - 1.234.567.891"
```

3.4.15 Conversioni da string

- Conversioni da string (in un Int32):

► `Int32.Parse(string str)`

```
Int32.Parse(" - 1234567891"); // - 1234567891
```

```
Int32.Parse(" - 1.234.567.891"); // FormatException
```

```
Int32.Parse(""); // FormatException
```

```

Int32.Parse(" - 1234567891999"); // OverflowException
Int32.Parse(null); // ArgumentNullException
► Int32.Parse(string str, System.Globalization.NumberStyles style)
NumberStyles determina lo stile permesso per i parametri stringa passati ai
metodi Parse delle classi basi numeriche

```

- I simboli da usare per la valuta, il separatore delle migliaia, il separatore decimale e il simbolo del segno sono specificati da NumberFormatInfo
- Gli attributi di NumberStyles sono indicati usando l'OR (inclusivo) bit-a-bit dei vari flag di campo

```

Int32.Parse(" - 1.234.567.891",
System.Globalization.NumberStyles.Number); // ok
Int32.Parse("B669FD2D", System.Globalization.NumberStyles.HexNumber); // ok

```

«enumeration» NumberStyles	
None = 0	
AllowLeadingWhite = 1	
AllowTrailingWhite = 2	
AllowLeadingSign = 4	
AllowTrailingSign = 8	
AllowParentheses = 16	
AllowDecimalPoint = 32	
AllowThousands = 64	
AllowExponent = 128	
AllowCurrencySymbol = 256	
AllowHexSpecifier = 512	
Integer = 7	
HexNumber = 515	
Number = 111	
Float = 167	
Currency = 383	
Any = 511	

3.4.16 Conversioni a/da string

- Conversioni a string (di un Int32):
 - Convert.ToString(int value, int toBase)
toBase= 2, 8, 10, 16


```

int k1 = -1234567891;
Convert.ToString(k1); // " - 1234567891"
Convert.ToString(k1,10); // " - 1234567891"
Convert.ToString(k1,16); // "b669fd2d"

```
- Conversioni da string (in un Int32):
 - Convert.ToInt32(string str, int fromBase)
fromBase= 2, 8, 10, 16


```

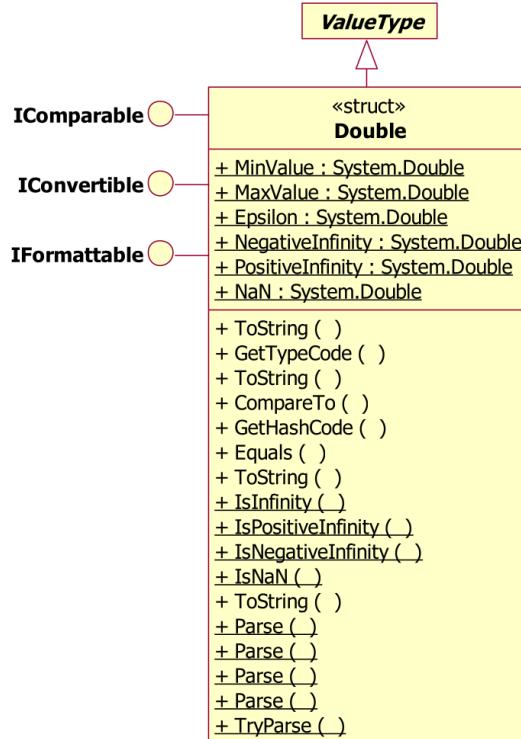
Convert.ToInt32("-1234567891"); // -1234567891
Convert.ToInt32("-1234567891",10); // -1234567891
Convert.ToInt32("B669FD2D",16); // -1234567891
Convert.ToInt32("0xB669FD2D",16); // -1234567891
Convert.ToInt32("B669FD2D",10); // FormatException

```

3.4.17 System.Double

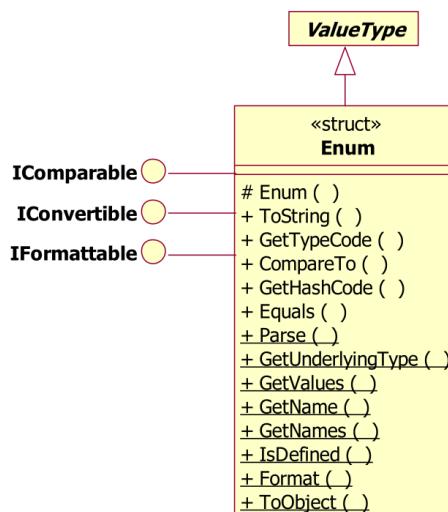
- Segue le specifiche IEEE 754
- Supporta ±0, ±Infinity, NaN
- Epsilon rappresenta il più piccolo Double positivo > 0

- Il metodo TryParse è analogo al metodo Parse, ma non lancia eccezioni in caso di fallimento della conversione
 - Se la conversione ha successo, il valore di ritorno è true e il parametro di uscita è posto pari al risultato della conversione
 - Se la conversione fallisce, il valore di ritorno è false e il parametro di uscita è posto pari a zero



3.4.18 System.Enum

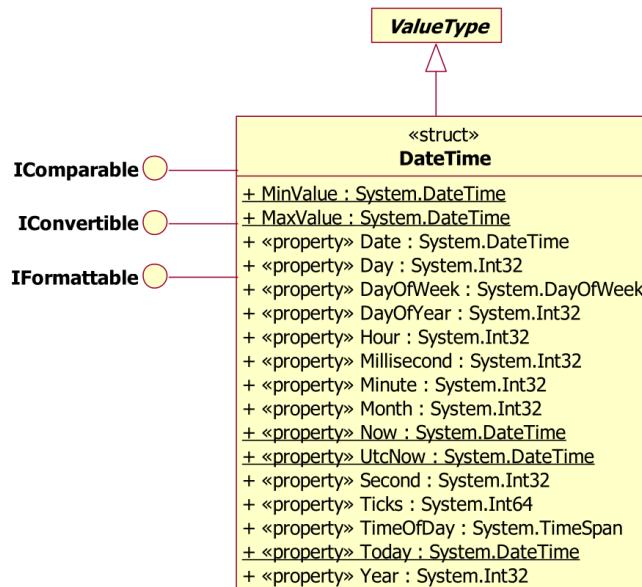
- Enum fornisce metodi per
 - Confrontare istanze di questa classe
 - Convertire il valore di un'istanza nella sua rappresentazione a stringa
 - Convertire la rappresentazione a stringa di un numero in un'istanza della classe
 - Creare un'istanza di un'enumerazione e valore specifico
- È possibile trattare un Enum come un campo bit



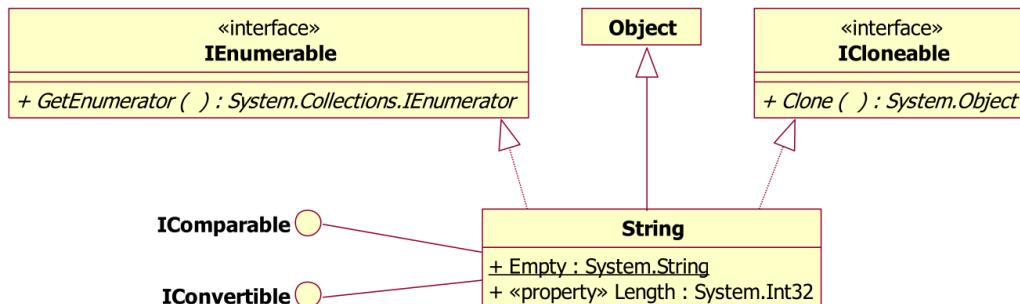
3.4.19 System.DateTime

- Rappresenta un istante nel tempo, tipicamente espresso come data e ora del giorno

- Il tipo di valore `DateTime` rappresenta le date e le ore con valori che vanno dalla mezzanotte del 1 gennaio 0001 d.C. (Era Comune) alle 11:59:59 P.M., 31 dicembre 9999 d.C.
- I valori temporali sono misurati in unità di 100 ns chiamate tick
- `DateTime` rappresenta un istante nel tempo, mentre `TimeSpan` rappresenta un intervallo di tempo

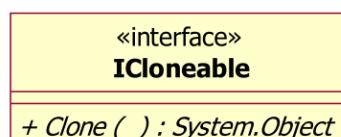


3.4.20 System.String



- Una stringa immutabile di caratteri Unicode a lunghezza
- Una `String` è detta immutabile perché, una volta creata, il suo valore non può essere modificato
- I metodi che sembrano modificare una `String`, in realtà restituiscono una nuova `String` contenente la modifica
- Se è necessario modificare il contenuto effettivo di un oggetto tipo stringa, utilizzare la classe `System.Text.StringBuilder`
- Per la classe `String` l'operatore di uguaglianza `==` è ridefinito, quindi, al contrario di Java, l'operatore ritorna `true` se le due stringhe che compara hanno lo stesso contenuto lessico-grafico

3.4.21 System.ICloneable



- Supporta la clonazione, che crea una nuova istanza di una classe con lo stesso stato di un'istanza esistente

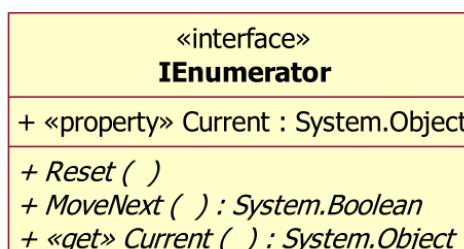
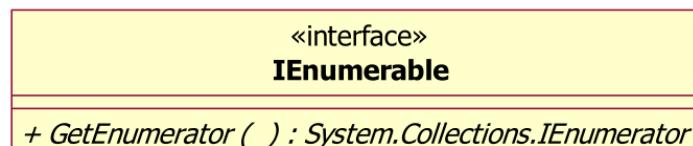
- Clone crea un nuovo oggetto che è una copia dell'istanza corrente
- Clone può essere implementato come:
 - una **copia superficiale** (shallow), vengono duplicati solo gli oggetti di primo livello, non vengono create nuove istanze di alcun campo

```
public object Clone()
{
    return MemberwiseClone();
}
```

- una **copia profonda** (deep), tutti gli oggetti vengono duplicati
- Clone restituisce una nuova istanza dello stesso tipo (o eventualmente un tipo derivato) dell'oggetto corrente

3.4.21.1 System.Collections.IEnumerable

- GetEnumerator restituisce un enumeratore che può essere utilizzato per scorrere una collezione
- Espone l'enumeratore, che supporta una semplice iterazione su una collezione



- Gli enumeratori permettono solamente di leggere i dati della collezione
- Gli enumeratori non possono essere usati per modificare la collezione
- Reset riporta l'enumeratore allo stato iniziale
- MoveNext si sposta all'elemento successivo, restituendo
 - true se l'operazione ha successo
 - false se l'enumeratore ha oltrepassato l'ultimo elemento
- Current restituisce l'oggetto attualmente referenziato

3.4.21.2 System.Collections.IEnumerator

- Non deve essere implementata direttamente da una classe contenitore
- Deve essere implementata da una classe separata (eventualmente annidata nella classe contenitore) che fornisce la funzionalità di iterare sulla classe contenitore
- Tale suddivisione di responsabilità permette di utilizzare contemporaneamente più enumeratori sulla stessa classe contenitore
- La classe contenitore deve implementare l'interfaccia **IEnumerable**
- Se una classe contenitore viene modificata, tutti gli enumeratori a essa associati vengono invalidati e non possono più essere utilizzati (**InvalidOperationException**)

```
IEnumerator enumerator = enumerable.GetEnumerator();
```

```
while (enumerator.MoveNext())
```

```

{
    MyType obj = (MyType) enumerator.Current;
    ...
}

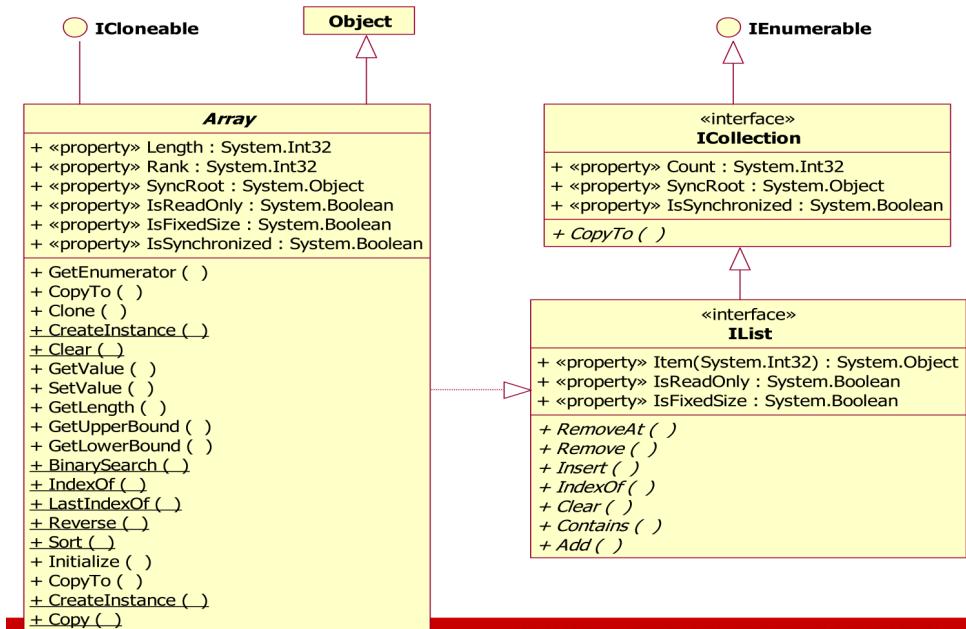
foreach (MyType obj in enumerable)
{
    ...
}

public class Contenitore : IEnumerable
{
    ...
    public IEnumerator GetEnumerator()
    {
        return new Enumeratore(this);
    }

    class Enumeratore : IEnumerator
    {
        public Enumeratore(Contenitore contenitore) ...
    }
}

```

3.4.22 System.Array



- Array mono-dimensionali

```

int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
// array of references
SomeClass[] d = new SomeClass[10];
// array of values (directly in the array)
SomeStruct[] e = new SomeStruct[10];

```

- Array multidimensionali (frastagliati): ogni array può avere dimensione diversa, per questo non possono essere inizializzati direttamente (“in un colpo solo”)

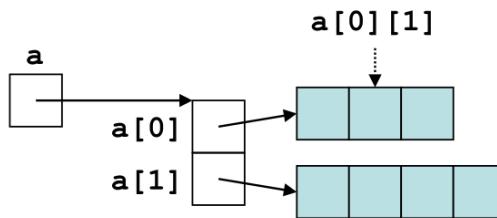
```
// array of references to other arrays
int[][] a = new int[2][];
// cannot be initialized directly
a[0] = new int[] {1, 2, 3};
a[1] = new int[] {4, 5, 6};
```

- Array multidimensionali (matrici)

```
// block matrix
int[,] a = new int[2, 3];
// can be initialized directly
int[,] b = {{1, 2, 3}, {4, 5, 6}};
int[, ,] c = new int[2, 4, 2];
```

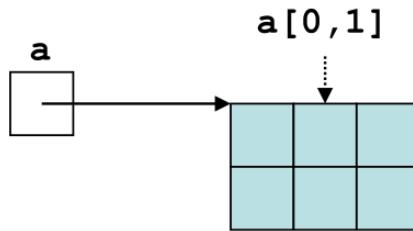
- **Frastagliati** (come in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];
...
int x = a[0][1];
```



- Matrici (come in C, più compatti ed efficienti)

```
int[,] a = new int[2, 3];
...
int x = a[0, 1];
```



3.5 Delegati ed eventi

3.5.1 Delegati

- Sono oggetti che possono contenere **il riferimento (type safe) a un metodo**, tramite il quale il metodo può essere invocato
- **Oggetti funzione (functor)**: oggetti che si comportano come una funzione (metodo)
- Simili ai puntatori a funzione del C/C++, ma *object-oriented* e molto più potenti
- Utilizzo standard: funzionalità di **callback**
 - **Elaborazione asincrona**
 - **Elaborazione cooperativa** (il chiamato fornisce una parte del servizio, il chiamante fornisce la parte rimanente - es. qsortin C)

- ▶ **Gestione degli eventi** (chi è interessato a un certo evento si registra presso il generatore dell'evento, specificando il metodo che gestirà l'evento)

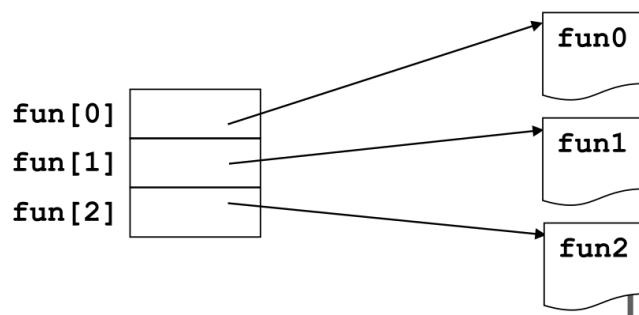
3.5.1.1 C/C++: Puntatori a funzioni

```
int funX(char c);
int funY(char c);
int (g)(char c) = NULL; //puntatore a funzione
...
g = cond1? funX : funY;
oppure: g = cond1? &funX : &funY;
...
... g('H') ... ≡ ... (g)('H') ...
```

3.5.1.2 C/C++: Array di puntatori a funzioni

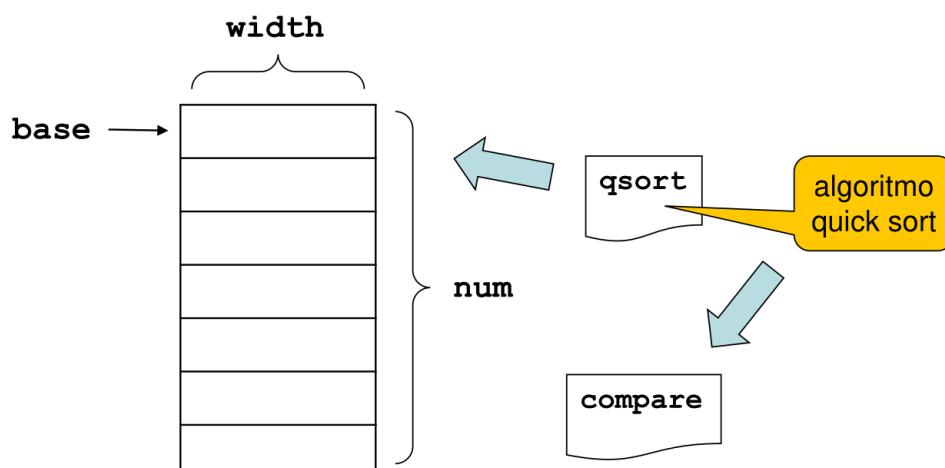
È possibile dichiarare un array di puntatori a funzione e invocare ogni singolo elemento dell'array o in sequenza tutti gli elementi dell'array

```
void fun0(char s);
void fun1(char s);
void fun2(char s);
void (fun[])(char s)
{ fun0, fun1, fun2 };
...
fun[m]("stringa di caratteri"); ≡
    (fun[m])("stringa di caratteri");
```



3.5.1.3 C/C++: Elaborazione cooperativa

```
void qsort(void base, int num, int width,
int (compare)(void , void ));
```



3.5.2 Delegati

- **Dichiarazione** di un nuovo **tipo di delegato** che può contenere il riferimento a un metodo che ha un unico argomento intero e restituisce un intero:
`delegate int Azione(int param);`
- **Definizione** di un **delegato**:

`Azione azione;`

- **Inizializzazione** di un delegato:

```
azione = new Azione(nomeMetodoStatico);
azione = new Azione(obj.nomeMetodo);
azione = nomeMetodoStatico; // C2.
azione = obj.nomeMetodo; // C2.
```

- **Invocazione del metodo** referenziato dal delegato:

```
int k1 = azione(10);
```

3.5.2.1 Delegati: Multicasting

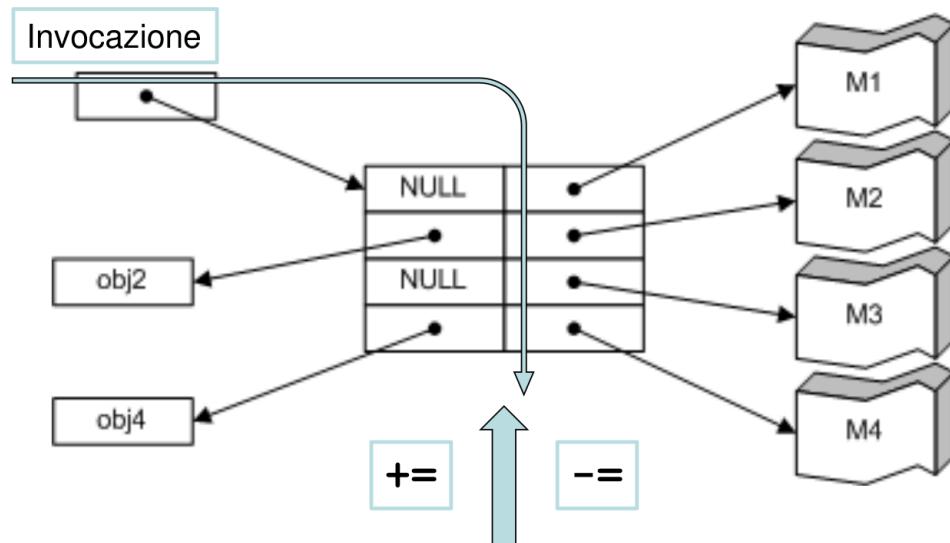
- È possibile assegnare al delegato una **lista di metodi** (invocation list)
- All'atto della chiamata del delegato, i metodi vengono invocati
 - **in sequenza**
 - **in modo sincrono**
- Per aggiungere un metodo alla lista: `+=`

```
Azione azione = Fun1;
... azione(10) ... // Fun1(10)
azione += Fun2;
... azione(10) ... // Fun1(10), Fun2(10)
```

- Per **togliere un metodo** dalla lista: `-=`

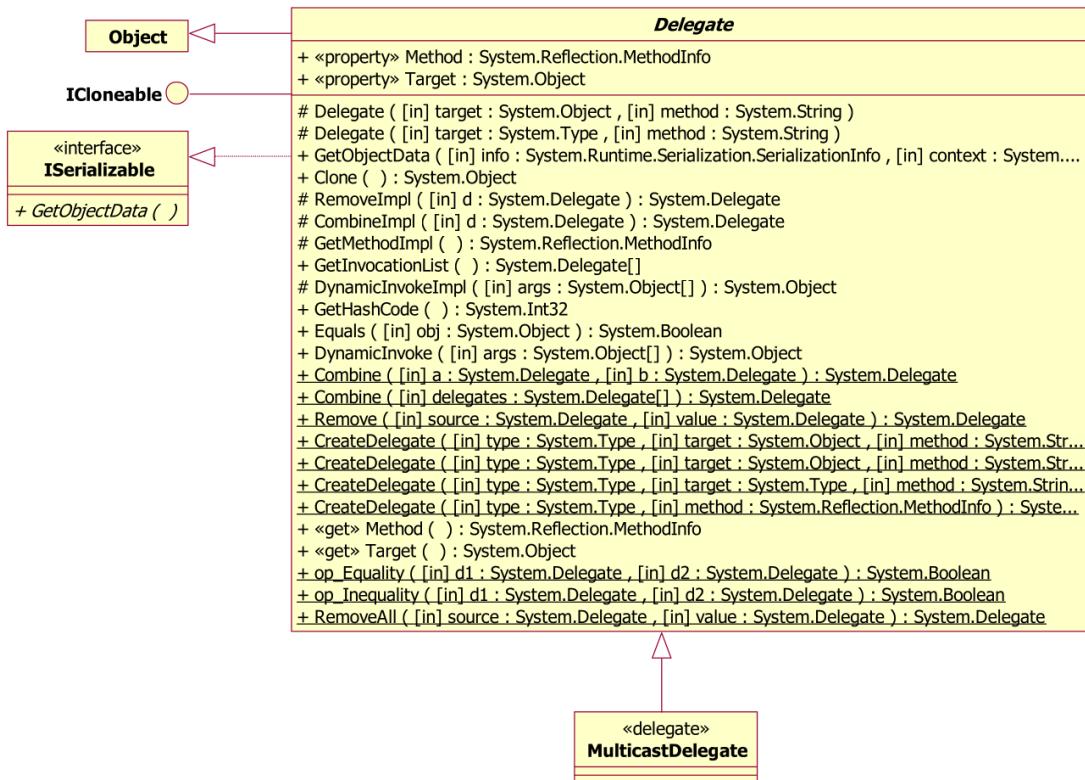
```
azione -= Fun1;
... azione(10) ... // Fun2(10)
```

- Un'istanza di delegato incapsula uno o più metodi (con una lista di parametri e tipo restituito specifici), ciascuno dei quali è indicato come entità invocabile (**callable entity**)
 - per i **metodi statici**, un'entità invocabile consiste solo in un metodo
 - per i **metodi di istanza**, un'entità invocabile consiste in un'istanza e un metodo su quell'istanza
- Un delegato
 - applica solo una singola firma del metodo (non un nome)
 - non conosce o non si preoccupa della classe dell'oggetto a cui fa riferimento
- Ciò rende i delegati utili per chiamate anonime (**anonymous invocation**)



- Possiamo vedere un *delegato* come un array di entità invocabili, formato da due coppie:
 - a destra il metodo
 - a sinistra l'istanza su cui quel metodo viene invocato (è NULL nel caso di metodi statici)

- L'invocazione di un'istanza del delegato la cui lista di metodi contiene più elementi procede richiamando ciascuno dei metodi nella lista, in modo sincrono, in ordine
- A ogni metodo così invocato viene passato lo stesso insieme di parametri fornito all'istanza del delegato
- Se tale chiamata di delegato include parametri di tipo riferimento
 - ogni invocazione di metodo avverrà con un riferimento alla stessa variabile
 - le modifiche a tale variabile da parte di un metodo nella lista saranno visibili ai metodi successivi in lista
- Se la chiamata del delegato include parametri di output o un valore di ritorno
 - il loro valore finale verrà dall'invocazione dell'ultimo delegato nell'elenco



- In C#, la dichiarazione di un nuovo tipo di delegato definisce automaticamente una nuova classe derivata dalla classe `System.MulticastDelegate`

```

System.Object
System.Delegate
System.MulticastDelegate
Azione

```

- Pertanto, sulle istanze di `Azione` è possibile invocare i metodi definiti a livello di classi di sistema

3.5.2.2 Esempio Boss-Worker

- È necessario modellare un'interazione tra due componenti
 - un **Worker** che effettua un'attività (o lavoro)
 - un **Boss** che controlla l'attività dei suoi Worker
- Ogni Worker deve notificare al proprio Boss:
 - quando il lavoro inizia
 - quando il lavoro è in esecuzione
 - quando il lavoro finisce
- Soluzioni possibili:
 1. class-based callback relationship
 2. interface-based callback relationship
 3. pattern Observer(lista di notifiche)
 4. delegate-based callback relationship
 5. event-based callback relationship

3.5.2.3 Una Relazione di Chiamata Basata su Delegati

- Un delegato è un'entità *type-safe* che si pone tra 1 caller e 0+ call target e che agisce come un'interfaccia con un solo metodo

```

interface IWorkerEvents
{

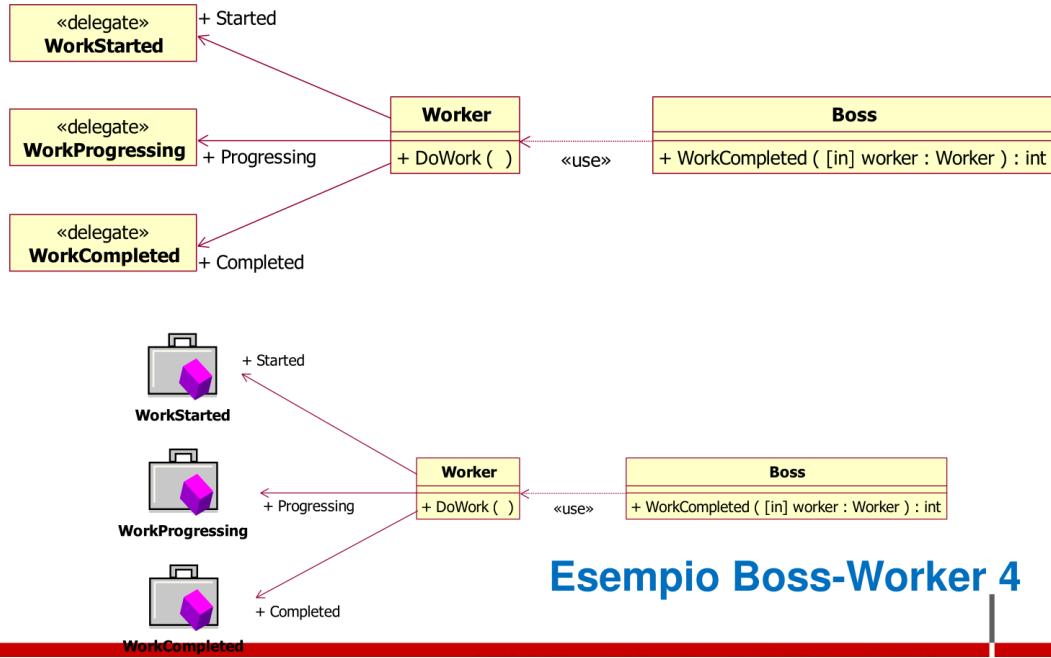
```

```

void WorkStarted(Worker worker);
void WorkProgressing(Worker worker);
int WorkCompleted(Worker worker);
}

delegate void WorkStarted(Worker worker);
delegate void WorkProgressing(Worker worker);
delegate int WorkCompleted(Worker worker);

```



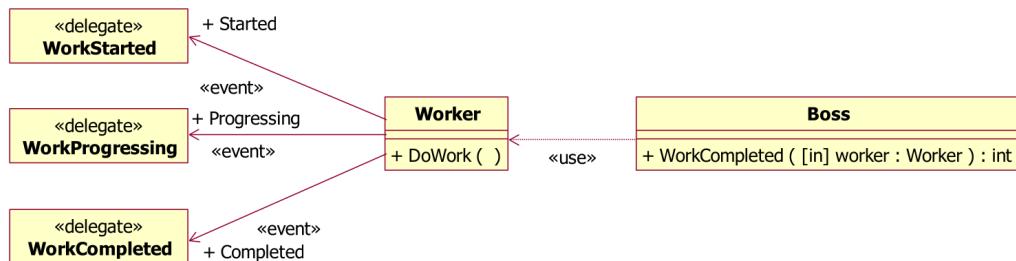
Dai Delegati agli Eventi

- L'utilizzo di campi pubblici per la registrazione fornisce un accesso eccessivo
 - I client possono sovrascrivere altri client precedentemente registrati
`peter.Started = WorkStarted;`
 - I client possono invocare i chiamati
`peter.Completed(peter);`
- Fornire metodi di registrazione pubblica abbinati al campo del delegato è una soluzione migliore, ma pesante se implementata manualmente
- Il modificatore `event` automatizza il supporto per
 - **public [un]registration**
 - **private implementation**

```

class Worker
{
    public event WorkStarted Started;
    public event WorkProgressing Progressing;
    public event WorkCompleted Completed;
    ...
}

```



Esempio Boss-Worker 4

Personalizzare la Registrazione a Eventi

- È possibile definire gestori di registrazione a eventi
 - Uno dei vantaggi di scrivere metodi propri di registrazione è l'aumentato controllo
 - La sintassi alternativa, analoga a una proprietà, supporta gestori di registrazione definiti dall'utente
 - Consente di rendere la registrazione condizionale o diversamente personalizzata
 - Sintassi di accesso lato client non modificata
 - È necessario fornire spazio per i client registrati

```

class Worker
{
    ...
    public event WorkProgressing Progressing
    {
        add
        {
            if(DateTime.Now.Hour < 12)
            { _progressing += value; }
            else
            { throw new InvalidOperationException
                ("Must register before noon."); }
        }
        remove
        { _progressing -= value; }
    }
    private WorkProgressing _progressing;
    ...
}

```

3.5.3 Eventi

- Evento**: “Fatto o avvenimento determinante nei confronti di una situazione oggettiva o soggettiva”
- In programmazione, un evento può essere scatenato
 - Dall’interazione con l’utente (click del mouse, ...)
 - dalla logica del programma

- **Event sender** - l'oggetto (o la classe) che scatena (raiseso triggers) l'evento (sorgente dell'evento)
- **Event receiver** - l'oggetto (o la classe) per il quale l'evento è determinante e che quindi desidera essere notificato quando l'evento si verifica (cliente)
- **Event handler** - il metodo (dell'*event receiver*) che viene eseguito all'atto della notifica
- Quando si verifica l'evento, il sender invia un messaggio di notifica a tutti i receiver
 - in pratica, invoca gli eventhandler di tutti i receiver
- In genere, il sender NON conosce né i *receiver*, né gli *handler*
- Il meccanismo che viene utilizzato per collegare sender e receiver/handler è il delegato (che permette **invocazioni anonime**)

3.5.3.1 Dichiarazione di un Evento - Convenzione .NET

- Un evento incapsula un delegato
 - è necessario dichiarare un tipo di delegato prima di poter dichiarare un evento
- Per convenzione, i delegati degli eventi in .NET hanno 2 parametri
 - la **sorgente** che ha scatenato l'evento e
 - i **dati** relativi all'evento
- Molti eventi, inclusi eventi della GUI come i click del mouse, non generano dati
- In tali situazioni, è sufficiente usare il delegato dell'evento fornito dalla libreria di classi per gli eventi senza dati, `System.EventHandler`
- Delegati di eventi personalizzati sono necessari solamente quando un evento genera dati

```
public delegate void EventHandler(object sender, EventArgs e);
```

```
System.Object
    System.Delegate
        System.MulticastDelegate
            System.EventHandler
```

- La classe `System.EventArgs` viene utilizzata quando un evento non deve passare informazioni aggiuntive ai propri gestori
- Se i gestori dell'evento hanno bisogno di informazioni aggiuntive, è necessario derivare una classe dalla classe `EventArgs`, aggiungere i dati necessari e utilizzare il delegate `EventHandler<TEventArgs>`

```
public event EventHandler Changed;
```

- In pratica, `Changed` è un delegato, ma la keyword `event` ne limita
 - la visibilità e
 - le possibilità di utilizzo
- Una volta dichiarato, l'evento può essere trattato come un delegato di tipo speciale
- In particolare, può:
 - essere `null` se nessun cliente si è registrato
 - essere associato a uno o più metodi da invocare

3.5.3.2 Invocazione di un Evento

- Per scatenare un evento è opportuno definire un metodo protetto virtuale `OnNomeEvento` e invocare sempre quello public event `EventHandler Changed` ; `protected virtual void OnChanged()`

```

{
    if ( Changed != null )
        Changed ( this , EventArgs.Empty );
}
...
OnChanged ();
...

```

- **Limitazione rispetto ai delegati**

L'invocazione dell'evento può avvenire solo all'interno della classe nella quale l'evento è stato dichiarato (benché l'evento sia stato dichiarato `public`)

- Al di fuori della classe in cui l'evento è stato dichiarato, un evento viene visto come un **delegato con accessi molto limitati**
- Le sole operazioni effettuabili dal cliente sono:
 - **agganciarsi a un evento**: aggiungere un nuovo delegato all'evento mediante l'operatore `+=`
 - **sganciarsi da un evento**: rimuovere un delegato dall'evento mediante l'operatore `-=`

3.5.3.2.1 Agganciarsi a un Evento

Per iniziare a ricevere le notifiche di un evento, il cliente deve:

- **Definire il metodo** (*event handler*) che dovrà essere invocato all'atto della notifica dell'evento (con la stessa signature dell'evento):

```
void ListChanged(object sender, EventArgs e)
{ ... }
```

- **Creare un delegato** dello stesso tipo dell'evento, farlo riferire al metodo e aggiungerlo alla lista dei delegati associati all'evento :

```
List.Changed += new EventHandler(ListChanged);
List.Changed += ListChanged; // C# 2.0
```

3.5.3.2.2 Sganciarsi da un Evento

Per smettere di ricevere le notifiche di un evento, il cliente deve:

- **Rimuovere il delegato** dalla lista dei delegati associati all'evento:

```
List.Changed -= new EventHandler ( ListChanged );
List.Changed -= ListChanged ; // C# 2.0
```

- Poiché `+=` e `-=` sono gli unici operatori permessi su un evento al di fuori del tipo che dichiara l'evento, il codice esterno al tipo
 - può aggiungere e rimuovere handler per un evento, ma
 - non può in nessun altro modo ottenere o modificare la lista di handlersottostante
- Gli eventi forniscono agli oggetti un modo utile per segnalare modifiche allo stato a clienti di tali oggetti
- Gli eventi sono un componente fondamentale **per la creazione di classi che possano essere riutilizzate in un gran numero di programmi differenti**

4 Principi di Design

4.1 Qualità della progettazione

- La **Qualità della progettazione** è un concetto vago
- La qualità dipende da specifiche **priorità dell' organizzazione**
- Un buon progetto potrebbe essere
 - il più affidabile,
 - il più efficiente,
 - il più manutenibile,
 - il più economico, ...
- Gli argomenti qui discussi riguardano principalmente la **manutenibilità del progetto**

4.2 Cosa rende un design “cattivo”?

- ✓ **Misdirection**: non soddisfa i requisiti
- ✓ **Rigidità del software**: una singola modifica influisce su molte altre parti del sistema
- ✓ **Fragilità del software**: una singola modifica influisce su parti inaspettate del sistema
- ✓ **Immobilità del software**: è difficile da riutilizzare in un'altra applicazione
- ✓ **Viscosità**: è difficile fare la cosa giusta, ma facile fare la cosa sbagliata

4.2.1 Rigidità del Software

- La tendenza per il software a essere **difficile da modificare**
- **Sintomo**: ogni modifica provoca una cascata di modifiche successive nei moduli dipendenti
- **Effetto**: i manager hanno timore ad accettare che gli sviluppatori risolvano problemi non critici - non sanno se/quando gli sviluppatori termineranno le modifiche

4.2.2 Fragilità del Software

- La tendenza del software a “**rompersi**” in molti punti ogni volta che viene modificato: i cambiamenti tendono a causare **comportamenti inaspettati** in altre parti del sistema (spesso in aree che non hanno alcuna relazione concettuale con l'area che è stata modificata)
- **Sintomo**: ogni correzione peggiora le cose, introducendo più problemi di quelli risolti: tale software è impossibile da manutenere
- **Effetto**: ogni volta che i manager autorizzano una correzione, temono che il software si “rompa” in modo inaspettato

4.2.3 Immobilità del Software

- L'**impossibilità di riutilizzare il software** di altri progetti o di parti dello stesso progetto
- **Sintomo**: uno sviluppatore scopre di aver bisogno di un modulo simile a quello scritto da un altro sviluppatore. Ma il modulo in questione ha troppe dipendenze. Dopo molto lavoro, lo sviluppatore scopre che il lavoro e il rischio necessari per

separare le parti desiderabili del software dalle parti indesiderabili sono troppo grandi per essere tollerati

- **Effetto:** e così il software viene semplicemente riscritto anziché riutilizzato

4.2.4 Viscosità del Software

- Gli sviluppatori di solito trovano più di un modo per apportare una modifica
 - alcuni preservano il design, altri no (cioè sono hack)
- La tendenza a **incoraggiare modifiche al software che sono hack** piuttosto che modifiche al software che preservano l'intento di progettazione originale
 - **Viscosità del design:** i metodi che preservano il design sono più difficili da utilizzare rispetto agli hack
 - **Viscosità dell'ambiente:** l'ambiente di sviluppo è lento e inefficiente (tempi di compilazione molto lunghi, il sistema di controllo dei sorgenti richiede ore per archiviare pochi file, ...)
- **Sintomo:** è facile fare la cosa sbagliata, ma difficile fare la cosa giusta, cioè è difficile fare modifiche se seguano il design indicato nel documento
- **Effetto:** la manutenibilità del software degenera a causa di hack, scorciatoie, correzioni temporanee, ...

4.2.5 Perché esistono risultati di progettazione scadenti?

- Ragioni ovvie:
 - mancanza di capacità/pratica di progettazione
 - tecnologie in evoluzione
 - vincoli di tempo/risorse
 - complessità del dominio, ...
- Non così ovvie:
 - **la “putrefazione” del software è un processo lento** - anche un design originariamente pulito ed elegante può degenerare nel corso dei mesi/anni
 - i **requisiti** spesso cambiano in modi che non erano stati previsti dal design (o dal progettista) originale
 - le **dipendenze** tra moduli non pianificate e improprie si insinuano: le dipendenze non vengono gestite

4.3 Modifiche ai Requisiti

- Come ingegneri del software, sappiamo benissimo che i requisiti cambiano
- In effetti, la maggior parte di noi si rende conto che il documento dei requisiti è il **documento più volatile** dell'intero progetto
- Se i nostri progetti falliscono a causa del costante arrivo di requisiti in continua evoluzione, **la colpa è della nostra progettazione**
- Dobbiamo in qualche modo trovare un modo per rendere i nostri progetti resistenti a tali cambiamenti e proteggerli dalla putrefazione

4.4 Gestione delle Dipendenze

- Ciascuno dei quattro sintomi sopra menzionati è causato (direttamente o indirettamente) da **dipendenze improprie tra moduli** del software
- È l'**architettura delle dipendenze** che si sta degradando e con essa la capacità del software di essere manutenuto
- Per prevenire il degrado dell'architettura delle dipendenze, è necessario gestire le dipendenze tra i moduli in un'applicazione

- La progettazione orientata agli oggetti è piena di **principi** e **tecniche** per la gestione delle dipendenze dei moduli

4.5 Principi di Design

- The **Single Responsibility Principle** (SRP)
- The **Dependency Inversion Principle** (DIP)
- The **Interface Segregation Principle** (ISP)
- The **Open/Closed Principle** (OCP)
- The **Liskov Substitution Principle** (LSP)

4.5.1 Premessa

4.5.1.1 Il principio zero

- Il principio zero è un principio di logica noto come **rasoio di Occam**:
“*Entia non sunt multiplicanda praeter necessitatem*”
ovvero: non bisogna introdurre concetti che non siano strettamente necessari
- È la forma “colta” di un principio pratico:
“*Quello che non c’è non si rompe*” (H. Ford)
- Tra due soluzioni bisogna preferire quella
 - che introduce il minor numero di ipotesi
 - che fa uso del minor numero di concetti

4.5.1.2 Semplicità e semplicismo

- La **semplicità** è un fattore importantissimo
 - il software deve fare i conti con una notevole componente di complessità, generata dal contesto in cui deve essere utilizzato
 - quindi è estremamente importante **non aggiungere altra complessità arbitraria**
- Il problema è che
 - la semplicità richiede uno **sforzo non indifferente** (è molto più facile essere **complicati che semplici**)
 - in generale le soluzioni più semplici vengono in mente per ultime
- Bisogna fare poi molta attenzione a essere semplici ma non semplicistici
“*Keep it as simple as possible but not simpler*” (A. Einstein)

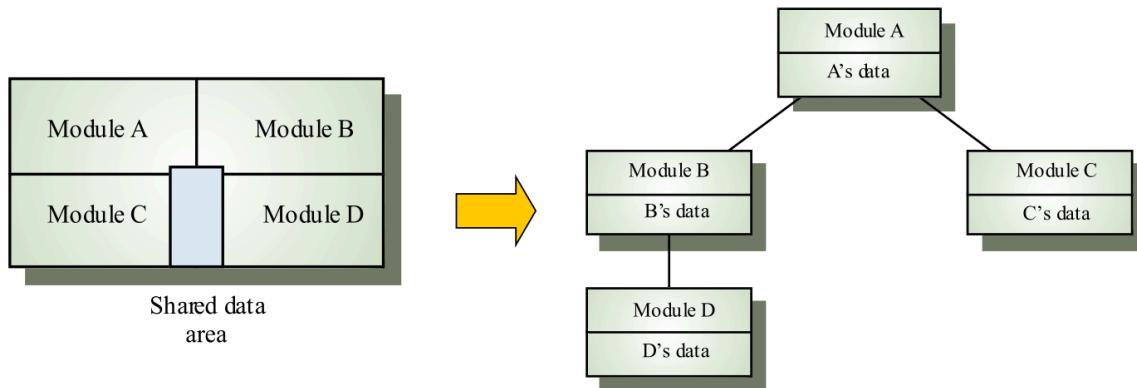
4.5.1.3 Divide et impera

- La **decomposizione** è una tecnica fondamentale per il controllo e la gestione della complessità
- Non esiste un solo modo per decomporre il sistema
 - la **qualità della progettazione** dipende direttamente dalla **qualità delle scelte di decomposizione** adottate
- In questo contesto il **principio fondamentale** è: minimizzare il grado di accoppiamento tra i moduli del sistema
- Da questo principio è possibile ricavare diverse regole:
 - minimizzare la quantità di interazione fra i moduli
 - eliminare tutti i riferimenti circolari fra moduli
 - ...

4.5.1.4 Rendere privati tutti i dati degli oggetti

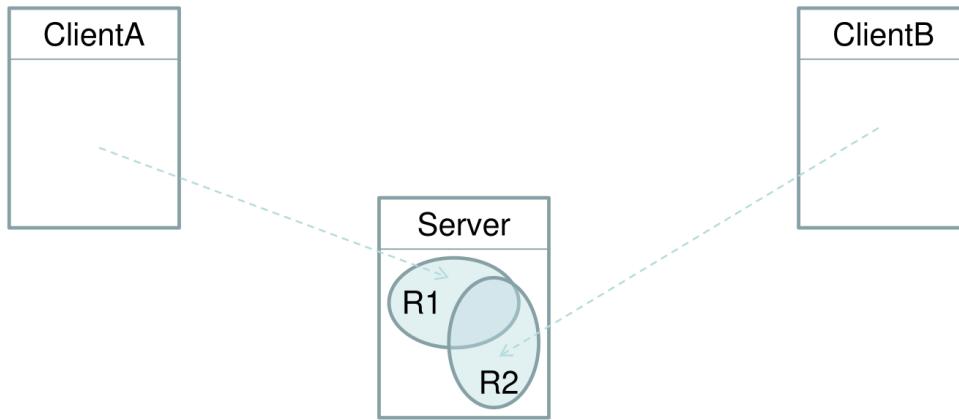
- Le modifiche ai dati pubblici rischiano sempre di “aprire” il modulo:

- ▶ possono avere un effetto domino che porta a richiedere modifiche in molte posizioni impreviste
- ▶ gli errori possono essere difficili da trovare e correggere completamente - le correzioni possono provocare errori altrove



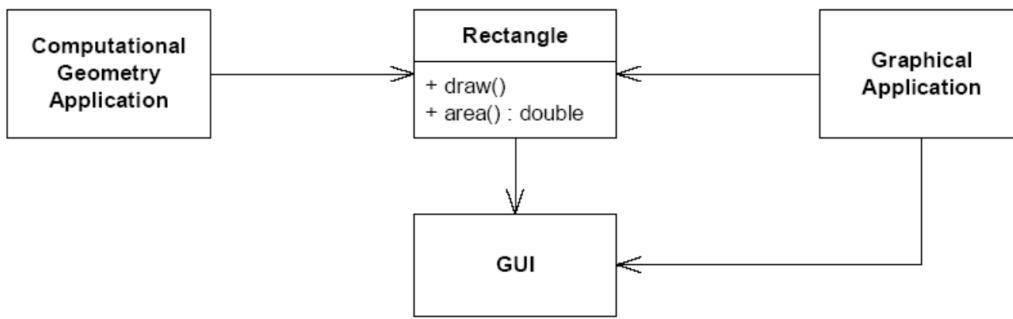
4.5.2 The Single Responsibility Principle

- *There should never be more than one reason for a class to change* (R. Martin)
- *A class has a single responsibility: it does it all, does it well, and does it only* (1-Responsibility Rule)
- Se una classe ha più di una responsabilità, queste diventano accoppiate
- Le modifiche a una responsabilità possono compromettere o inibire la capacità della classe di realizzare le altre
- Questo tipo di accoppiamento porta a **design fragili** che si rompono in modi inaspettati quando modificati



Una modifica di R1 può avere delle conseguenze anche su ClientB che NON utilizza direttamente R1

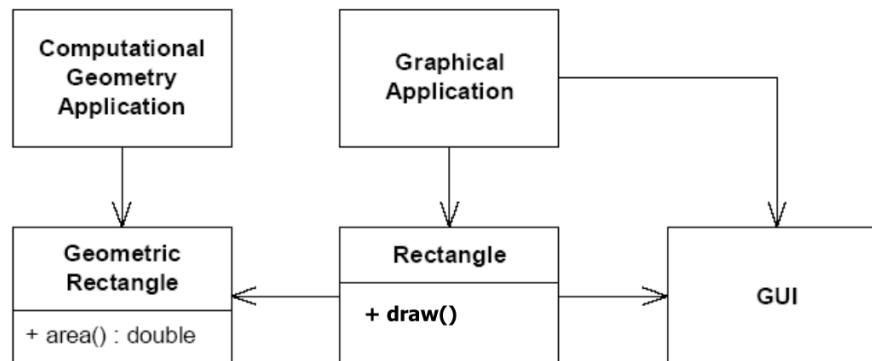
4.5.2.1 Esempio



- Un'applicazione è di geometria computazionale
 - usa Rectangle per aiutarsi con la matematica delle forme geometriche
 - non disegna mai il rettangolo sullo schermo
- L'altra applicazione è di natura grafica
 - può anche fare un po' di geometria computazionale, ma
 - disegna sicuramente il rettangolo sullo schermo
- La classe Rectangle ha **due responsabilità**:
 - la prima responsabilità è fornire un modello matematico della geometria di un rettangolo
 - la seconda responsabilità è disegnare il rettangolo su un'interfaccia grafica

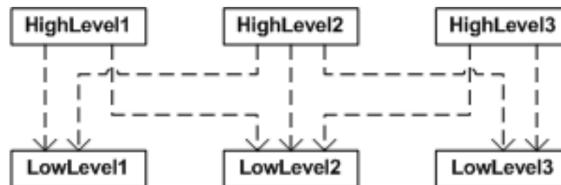
4.5.2.2 Esempio - Refactoring

- Un progetto migliore consiste nel **separare le due responsabilità in due classi completamente distinte**
- **Si estraе una classe**: si crea una nuova classe e si spostano i campi e i metodi opportuni dalla vecchia classe alla nuova classe

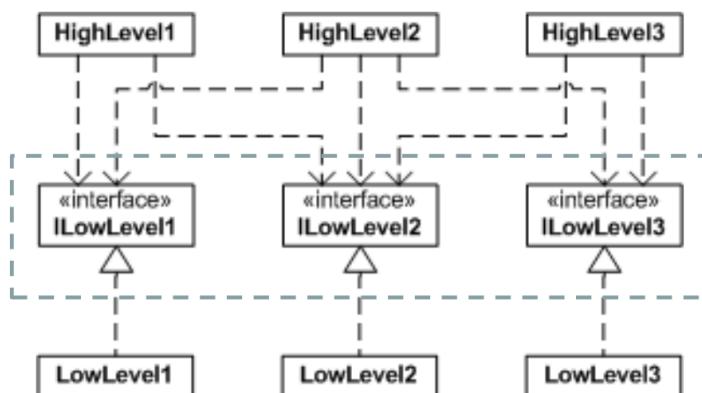
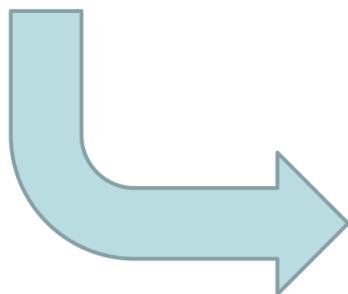


4.5.3 The Dependency Inversion Principle

- **Depend upon abstractions**
Do not depend upon concretions
 - Ogni dipendenza dovrebbe puntare a un'interfaccia o a una classe astratta
 - Nessuna dipendenza dovrebbe puntare a una classe concreta
 - I moduli di alto livello (i clienti) non dovrebbero dipendere dai moduli di basso livello (i fornitori di servizi)
- Entrambi dovrebbero dipendere da astrazioni



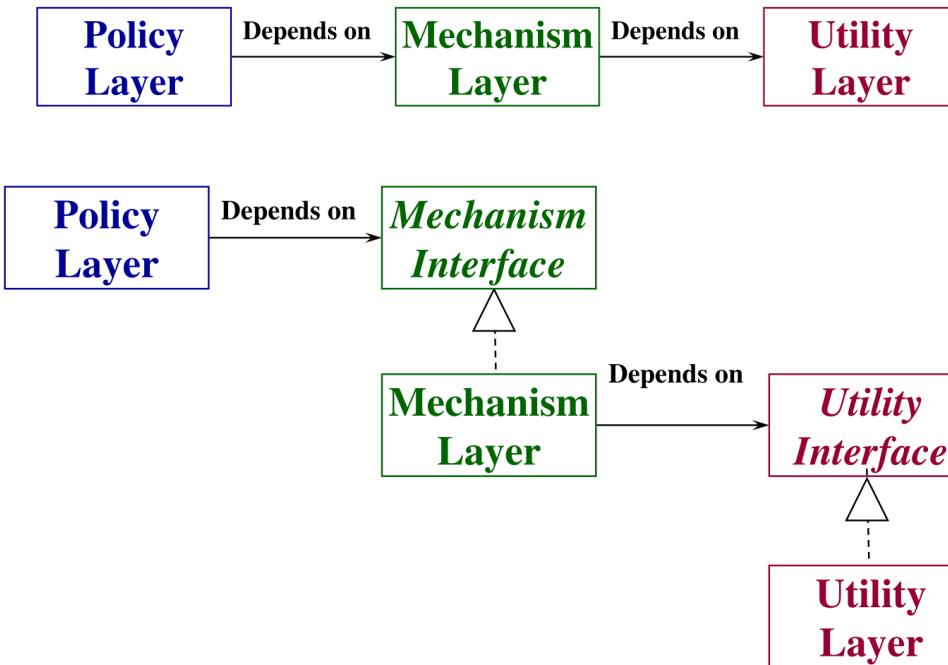
- I **moduli di basso livello** contengono la maggior parte del codice e della logica implementativa e quindi **sono i più soggetti a cambiamenti**
- Se i **moduli di alto livello** dipendono dai dettagli dei moduli di basso livello (sono accoppiati in modo troppo stretto), i cambiamenti si propagano e le conseguenze sono:
 - Rigidità**: bisogna intervenire su un numero elevato di moduli
 - Fragilità**: si introducono errori in altre parti del sistema
 - Immobilità**: i moduli di alto livello non si possono riutilizzare perché non si riescono a separare da quelli di basso livello
- Soluzione:
 - i moduli di basso livello implementano un'intefaccia
 - i moduli di alto livello utilizzano quell'intefaccia



- Questo principio funziona perché:
 - le astrazioni** contengono pochissimo codice (in teoria nulla) e quindi **sono poco soggette a cambiamenti**
 - i **moduli non astratti sono soggetti a cambiamenti** ma questi cambiamenti sono sicuri perché nessuno dipende da questi moduli
- I dettagli del sistema sono stati isolati, separati da un **muro di astrazioni stabili**, e questo impedisce ai cambiamenti di propagarsi (**design for change**)
- Nel contempo i singoli moduli sono **maggiormente riusabili** perché sono disaccoppiati fra di loro (**design for reuse**)

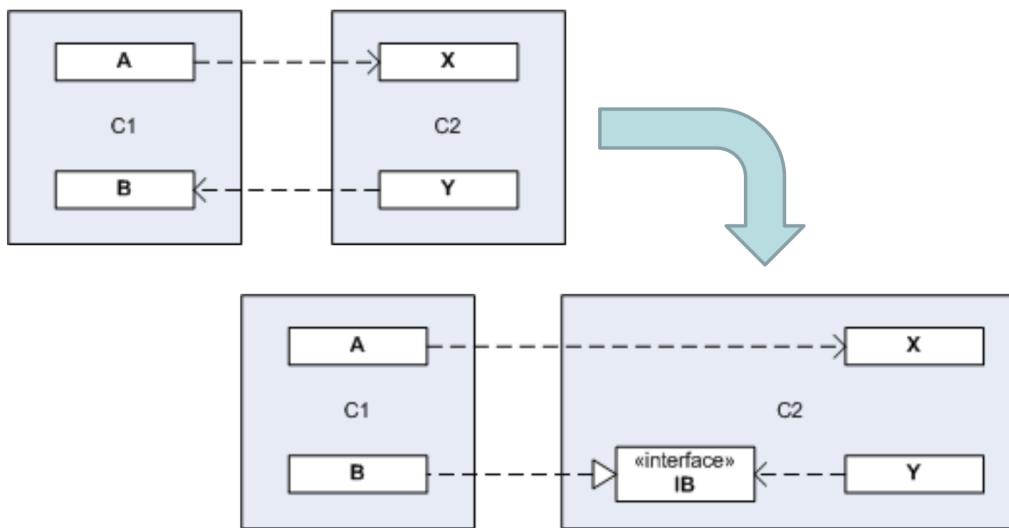
4.5.3.1 Dipendenze transitive

- “...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface” (Grady Booch)
- I sistemi software dovrebbero essere stratificati, cioè organizzati a livelli
- Le **dipendenze transitive** devono essere eliminate

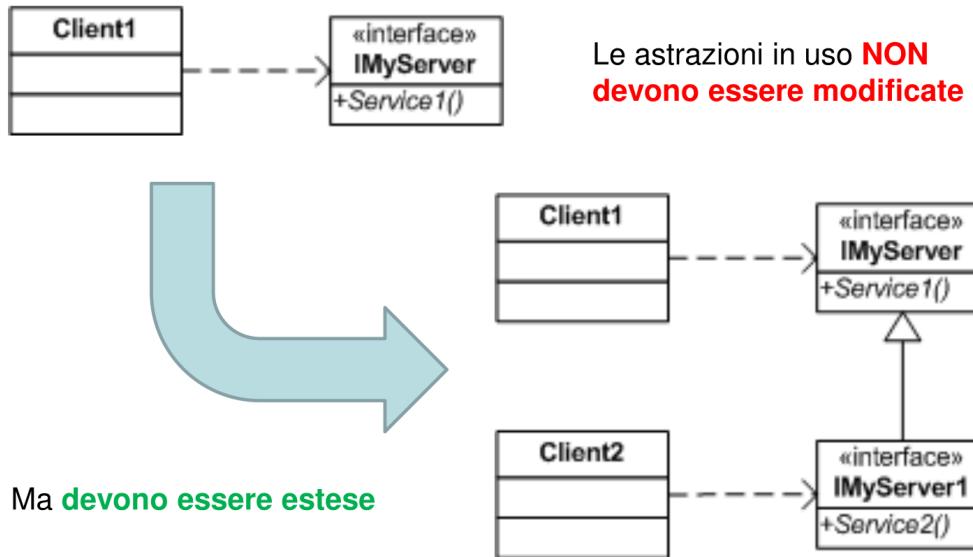


4.5.3.2 Dipendenze cicliche

- Le **dipendenze cicliche** devono essere eliminate



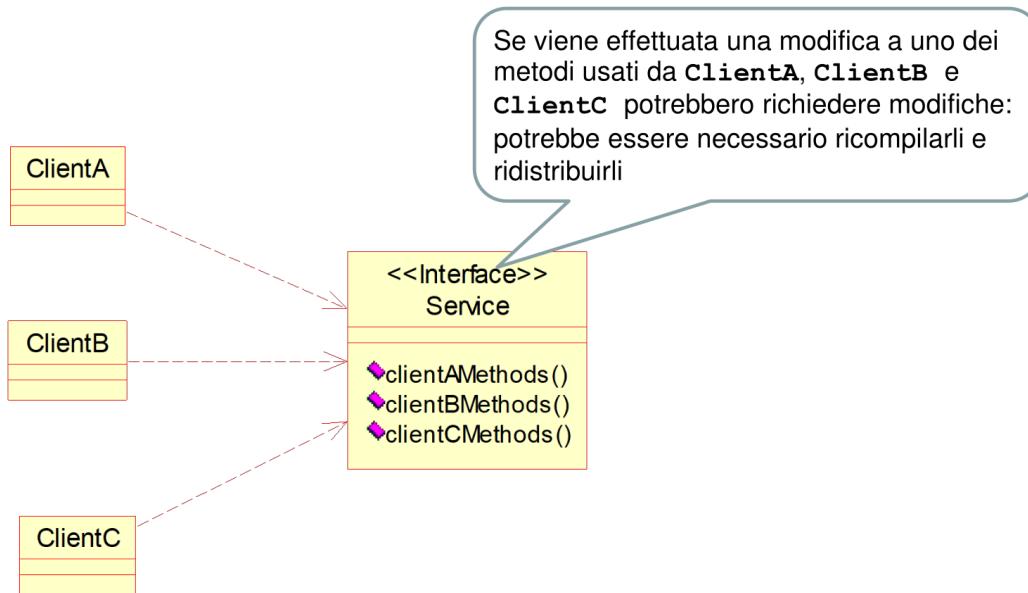
4.5.3.3 Stabilità delle astrazioni



4.5.4 The Interface Segregation Principle

- **Clients should not be forced to depend upon interfaces that they do not use**
- Molte interfacce specifiche per un cliente sono meglio di un'unica interfaccia general purpose

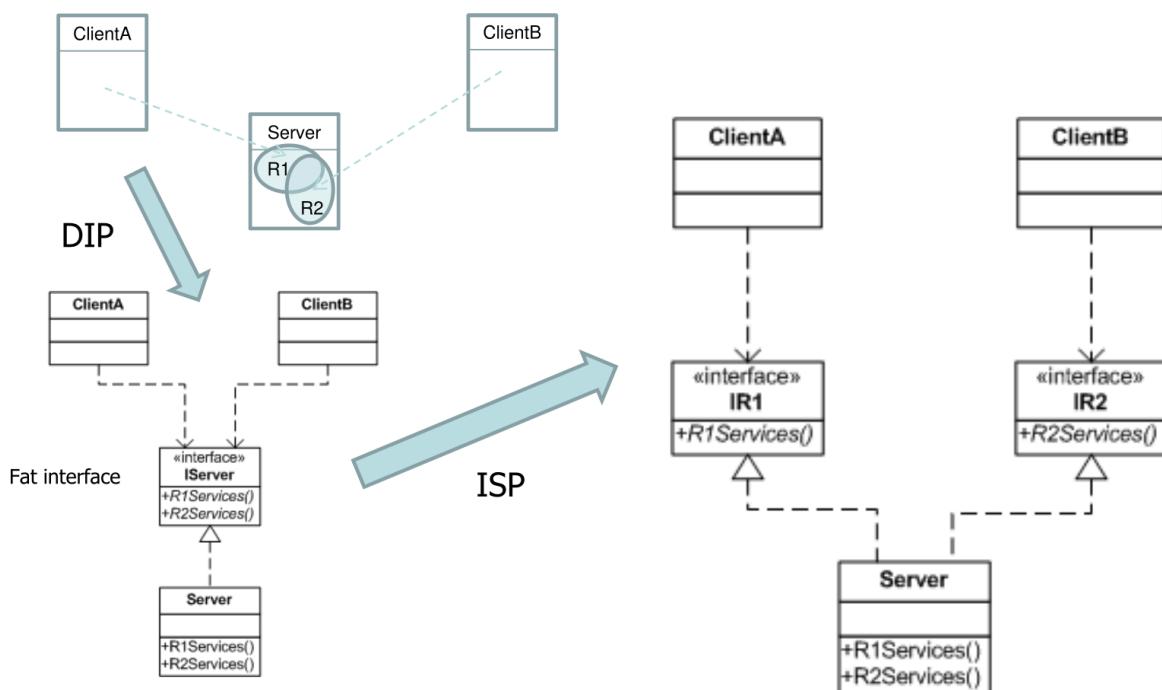
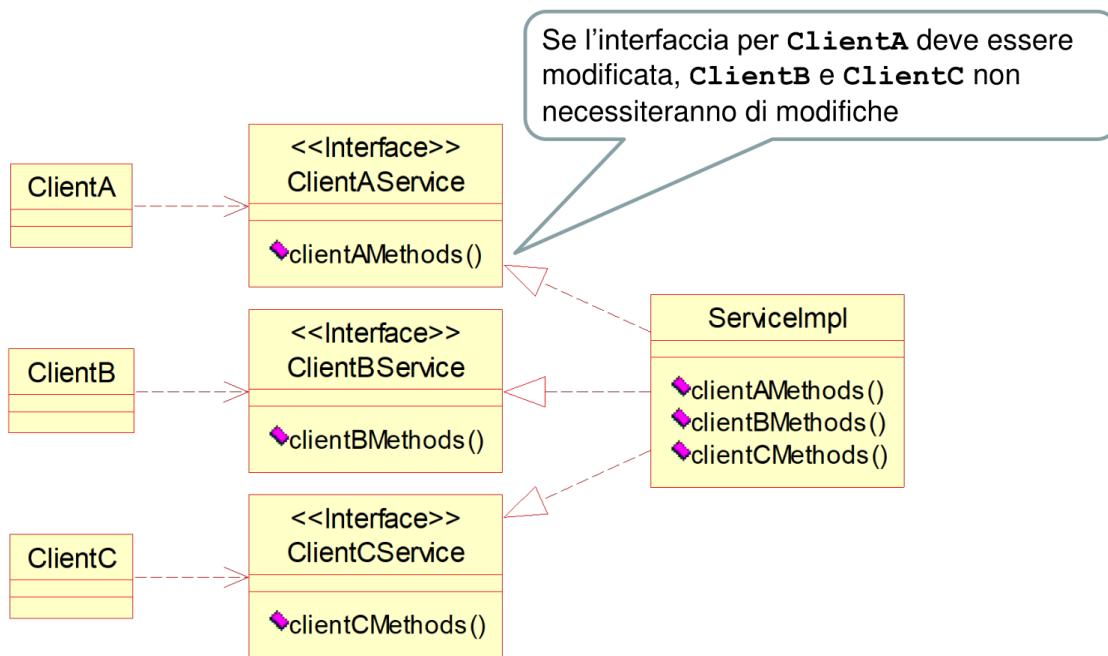
4.5.4.1 Fat Interface



4.5.5 The Interface Segregation Principle

- I clienti non dovrebbero dipendere da servizi che non utilizzano
- Le *fat interface* creano una forma indiretta di accoppiamento (inutile) fra i clienti - se un cliente richiede l'aggiunta di una nuova funzionalità all'interfaccia, ogni altro cliente è costretto a cambiare anche se non è interessato alla nuova funzionalità
- Questo crea un'inutile sforzo di manutenzione e può rendere difficile trovare eventuali errori
- Se i servizi di una classe possono essere suddivisi in gruppi e ogni gruppo viene utilizzato da un diverso insieme di clienti, creare interfacce specifiche per ogni tipo di cliente e implementare tutte le interfacce nella classe

4.5.5.1 Segregated Interfaces



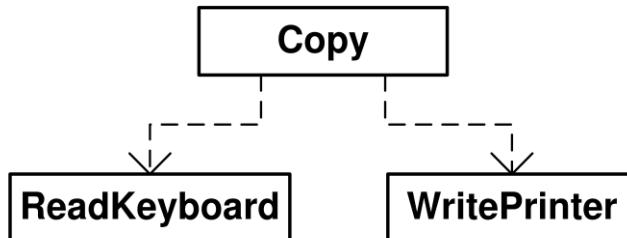
4.5.6 The Open/Closed Principle

- **Il principio più importante per la progettazione di entità riutilizzabili**
- *Software entities (classes, modules, functions , ...) should be open for extension, but closed for modification*
- **Open:**
 - **Possono essere estese** aggiungendo nuovo stato o proprietà comportamentali
- **Closed:**
 - Hanno un'interfaccia ben-definita, pubblica e stabile che **non può essere cambiata**
- Dobbiamo scrivere i moduli in modo che
 - **possano essere estesi**,
 - **senza la necessità di essere modificati**
- In altre parole, vogliamo
 - cambiare quello che fanno i moduli

- ▶ senza cambiare il codice dei moduli
- Apparentemente si tratta di una **contraddizione**: come può un modulo immutabile esibire un comportamento che non sia fisso nel tempo?
- La risposta risiede **nell'astrazione** : è possibile creare astrazioni che rendono un modulo immutabile, ma rappresentano un gruppo illimitato di comportamenti
- Il segreto sta nell'utilizzo di **interfacce** (o di **classi astratte**)
- A un'interfaccia **immutabile** possono corrispondere innumerevoli classi concrete che realizzano comportamenti diversi
- Un modulo che utilizza astrazioni
 - ▶ non dovrà mai essere modificato, dal momento che le astrazioni sono immutabili (**il modulo è chiuso per le modifiche**)
 - ▶ potrà cambiare comportamento, se si utilizzano nuove classi che implementano le astrazioni (**il modulo è aperto per le estensioni**)

4.5.6.1 Esempio 1

- Consideriamo un semplice programma che si occupa di copiare su una stampante i caratteri digitati su una tastiera
- Assumiamo, inoltre, che la piattaforma di implementazione non possieda un sistema operativo in grado di supportare l'indipendenza dal dispositivo



```

void Copy()
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
        WritePrinter(c);
}
  
```

Copy
ReadKeyboard WritePrinter

- **I due moduli di basso livello sono riutilizzabili**: possono essere usati in tanti altri programmi per accedere alla tastiera e alla stampante - è la stessa riusabilità offerta dalle librerie di classi
- **Il modulo “Copy” non è riutilizzabile** in un qualsiasi contesto che non includa una tastiera o una stampante
- È un peccato, perché tale modulo contiene “l'intelligenza del sistema”
 - ▶ è il modulo “Copy” che incapsula la funzionalità cui siamo interessati per il riuso
- Consideriamo un nuovo programma che copi caratteri da tastiera a un file su disco

- Potremmo modificare il modulo “Copy” per fornirgli questa nuova funzionalità
- Con l’andar del tempo, più e più dispositivi verranno aggiunti a questo programma di copia, e il modulo “Copy” sarà tappezzato di istruzioni if/else, diventando dipendente da diversi moduli di più basso livello
 - alla fine diverrà **rigido** e **fragile**

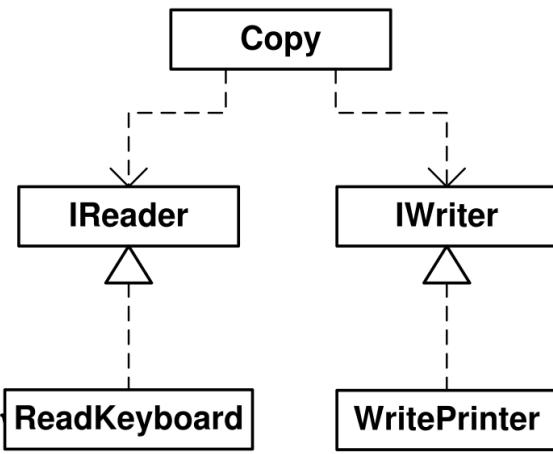
```
enum OutputDevice
{
    Printer,
    Disk
};
void Copy(OutputDevice dev)
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
    {
        if (dev == Printer)
            WritePrinter(c);
        else
            WriteDisk(c);
    }
}
```

- Un modo per caratterizzare il problema visto in precedenza è di notare che il modulo che contiene la politica di alto livello (Copy) dipende dai moduli di dettaglio e di più basso livello che controlla (WritePrinter ReadKeyboard)
- Se potessimo trovare un modo di rendere il modulo Copy indipendente dai dettagli che controlla, allora
 - potremmo **riutilizzarlo** liberamente
 - potremmo produrre altri programmi che usano questo modulo per **copiare caratteri da un qualsiasi dispositivo di input a un qualsiasi dispositivo di output**

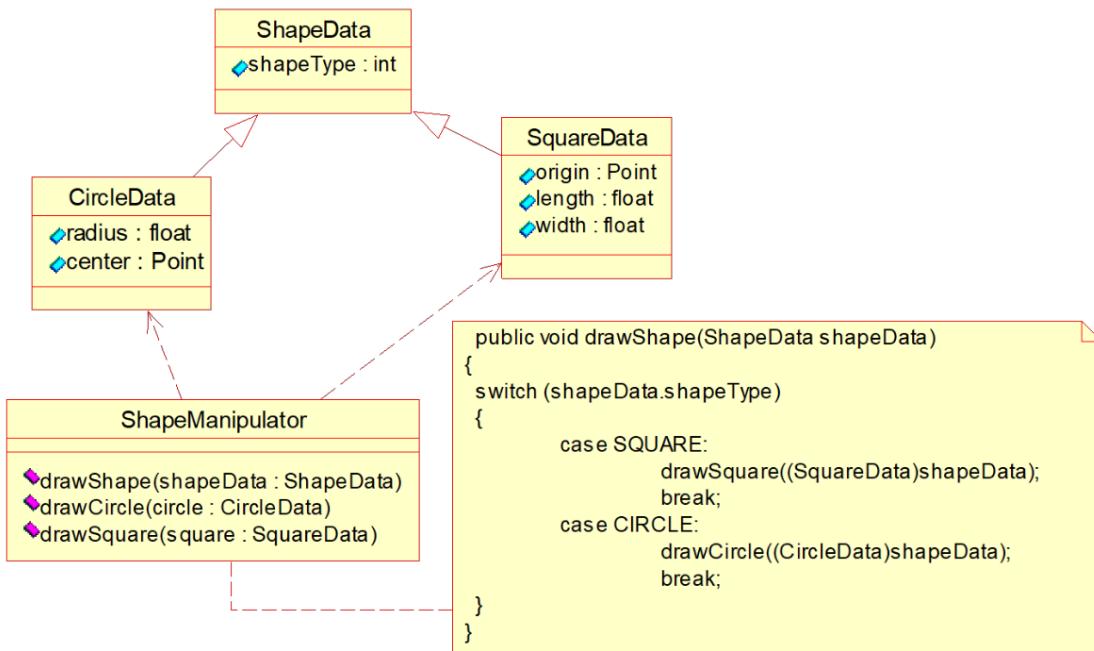
```
interface IReader
{
    int Read();
}

interface IWriter
{
    void Write(char);
}

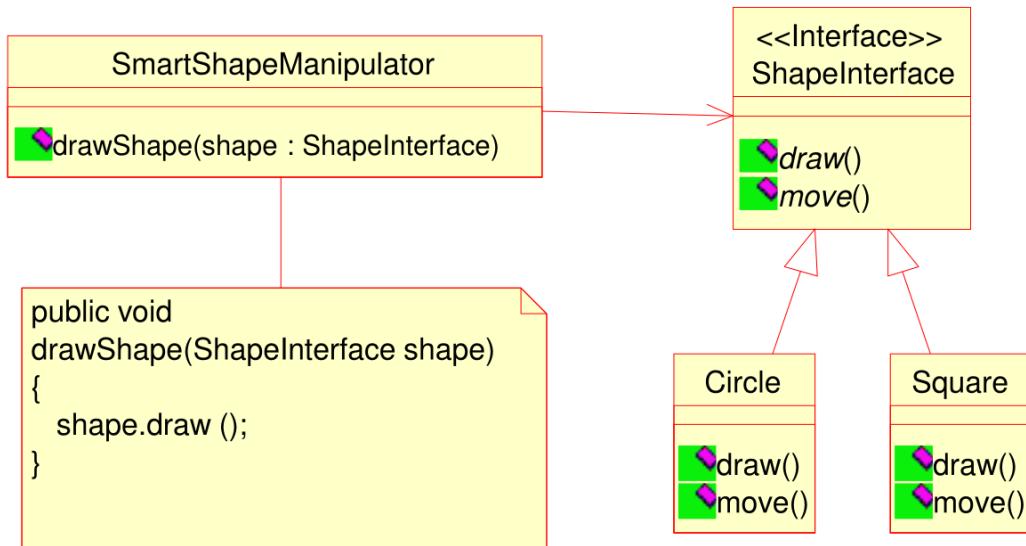
void Copy(IReader r, IWriter w)
{
    int c;
    while ((c = r.Read()) != EOF)
        w.Write(c);
}
```



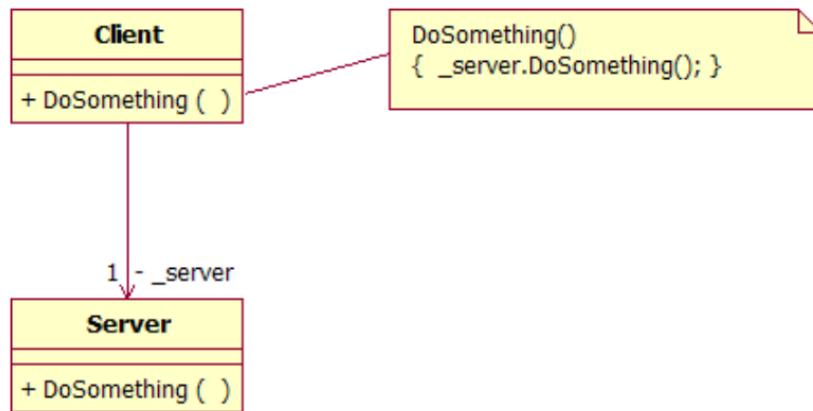
4.5.6.2 Esempio 2



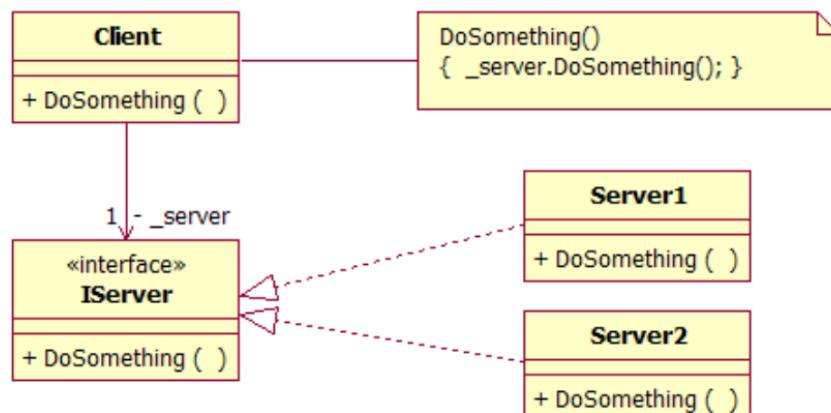
- **Se dovessi creare una nuova** shape, come Triangle, **dovrei modificare** drawShape
- In un'applicazione complessa l'istruzione switch/case verrebbe ripetuta più e più volte per ogni operazione possa essere effettuata su una shape
- Ancor peggio, ogni modulo che contenesse una tale istruzione switch/case statement **manterebbe una dipendenza da qualsiasi shape** possa essere disegnata
 - Quindi, ogni volta una singola shape dovesse essere modificata in un qualsiasi modo, tutti i moduli dovrebbero essere ricompilati ed eventualmente modificati



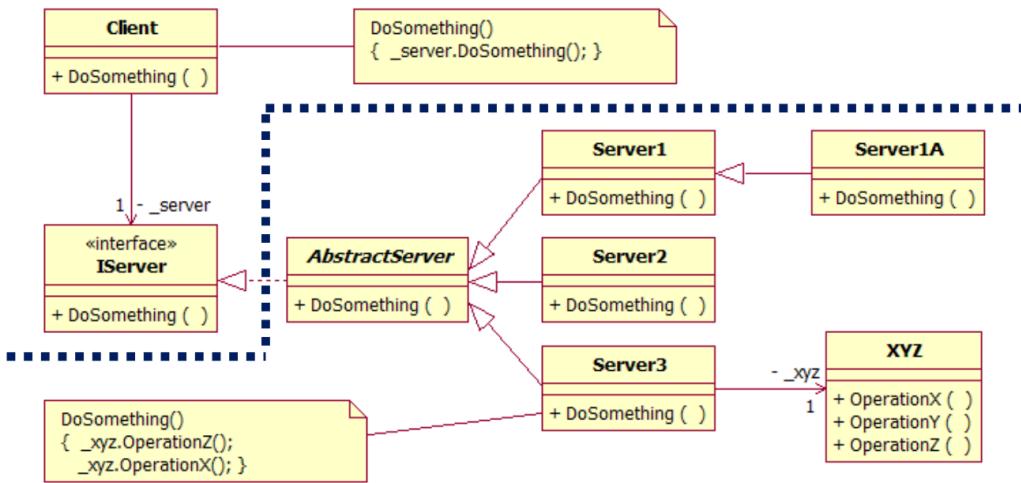
4.5.6.3 Esempio 3



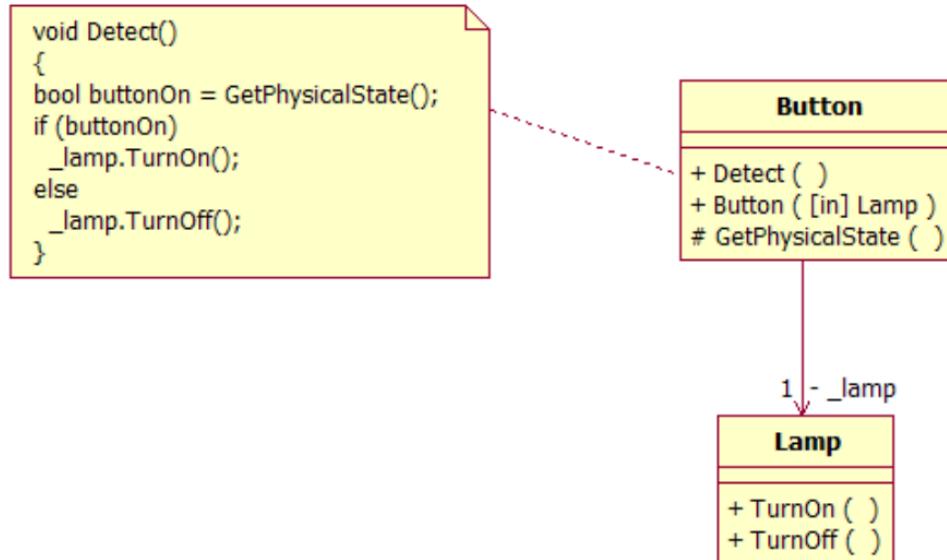
- Supponiamo di dover utilizzare un nuovo tipo di server!



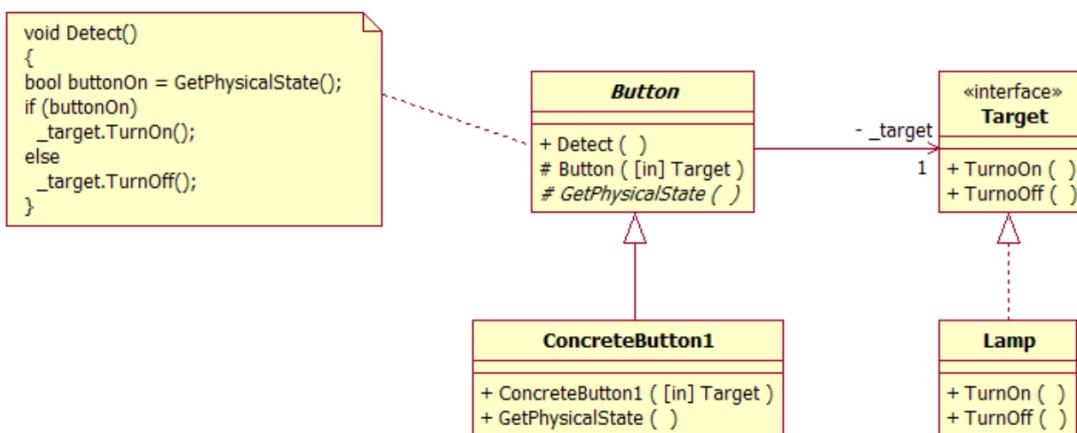
- Client è chiuso alle modifiche dell'implementazione di IServer
- Client è aperto all'estensione tramite nuove implementazioni di IServer
- Senza IServer , Client sarebbe aperto alle modifiche di Server



4.5.6.4 Esempio 4



- E se volessimo accendere un motore?



4.5.6.5 Conclusioni

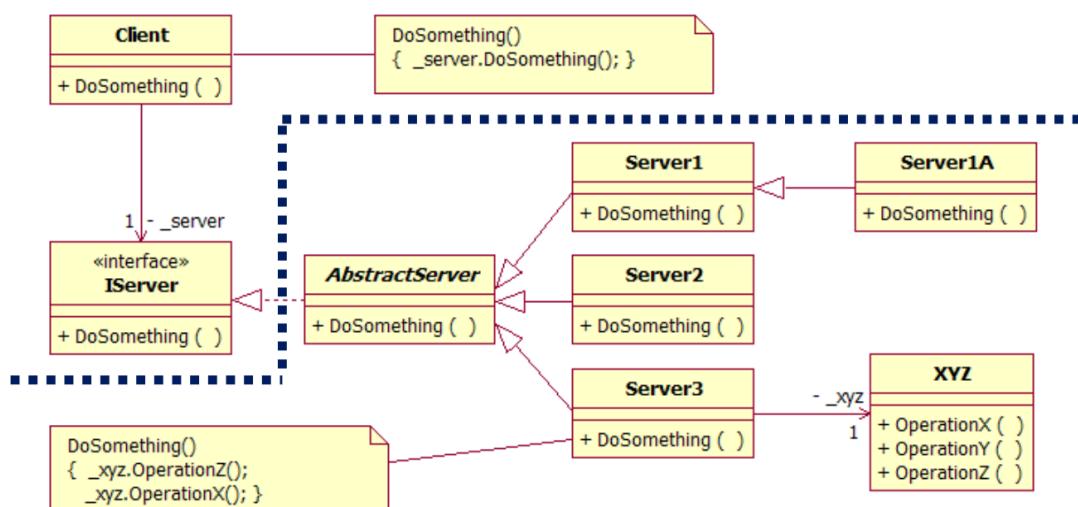
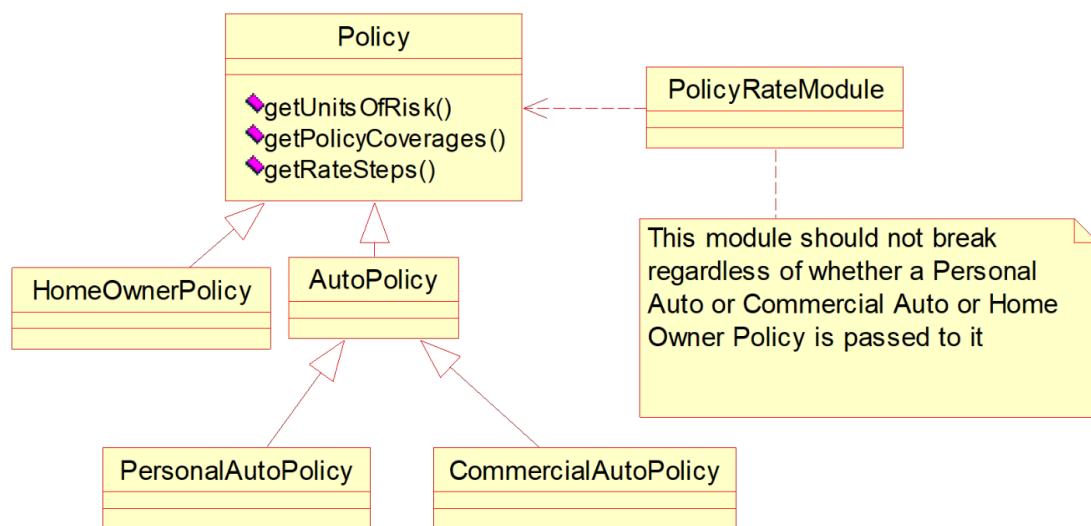
- Se la maggior parte dei moduli di un'applicazione segue OCP, allora
 - è possibile aggiungere nuove funzionalità all'applicazione
 - **aggiungendo nuovo codice**
 - invece che **cambiando codice funzionante**

- il codice che già funziona non è esposto a rotture

4.5.7 The Liskov Substitution Principle

- *Subclasses should be substitutable for their base classes* (Barbara Liskov)
- *All derived classes must honor the contracts of their base classes* (Design by Contract - Bertrand Meyer)
- Il cliente di una classe base deve continuare a funzionare correttamente se gli viene passato un sottotipo di tale classe base
- In altre parole: un cliente che usa istanze di una classe A deve poter usare istanze di una qualsiasi sottoclasse di A **senza accorgersi della differenza**

4.5.7.1 Example



- OCP si basa sull'uso di classi concrete derivate da un'astrazione (interfaccia o classe astratta)
- LSP costituisce una guida per creare queste classi concrete mediante l'ereditarietà
- La principale causa di violazioni al principio di Liskov è dato dalla **ridefinizione di metodi virtuali** nelle classi derivate: è qui che bisogna riporre la massima attenzione
- La chiave per evitare le violazioni al principio di Liskov risiede nel **Design by Contract** (B. Meyer)

4.6 Design by Contract

- Nel Design by Contract ogni metodo ha
 - un insieme di **pre-condizioni** - requisiti minimi che devono essere soddisfatti dal chiamante perché il metodo possa essere eseguito correttamente
 - un insieme di **post-condizioni** - requisiti che devono essere soddisfatti dal metodo nel caso di esecuzione corretta
- Questi due insiemi di condizioni costituiscono **un contratto** tra chi invoca il metodo (cliente) e il metodo stesso (e quindi la classe a cui appartiene)
 - le **pre-condizioni vincolano il chiamante**
 - le **post-condizioni vincolano il metodo**
 - se il chiamante garantisce il verificarsi delle pre-condizioni, il metodo garantisce il verificarsi delle post-condizioni
- Quando un metodo viene ridefinito in una sottoclass
 - le **pre-condizioni** devono essere **identiche o meno stringenti**
 - le **post-condizioni** devono essere **identiche o più stringenti**
- Questo perché un cliente che invoca il metodo conosce il contratto definito a livello della classe base, quindi non è in grado:
 - di soddisfare pre-condizioni più stringenti o
 - di accettare post-condizioni meno stringenti
- In caso contrario, il cliente dovrebbe conoscere informazioni sulla classe derivata e questo porterebbe a una violazione del principio di Liskov

```
public class BaseClass
{
    public virtual int Calculate(int val)
    {
        Precondition(-10000 <= val && val <= 10000);
        int result = val / 100;
        Postcondition(-100 <= result && result <= 100);
        return result;
    }
}

public class SubClass : BaseClass
{
    public override int Calculate(int val)
    {
        Precondition(-20000 <= val && val <= 20000);
        int result = Math.Abs(val) / 200;
        Postcondition(0 <= result && result <= 100);
        return result;
    }
}
```

4.7 Il Quadrato è un Rettangolo?

```
public class Rectangle
{
    private double _width;
    private double _height;
```

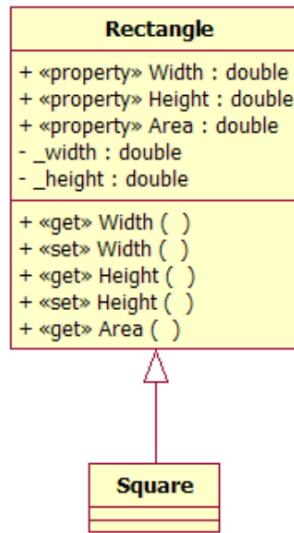
```

public double Width
{
    get { return _width; }
    set { _width = value; }
}

public double Height
{
    get { return _height; }
    set { _height = value; }
}
public double Area
{
    get { return Width * Height; }
}
}

```

- Immaginiamo che questa applicazione funzioni correttamente e sia installata in diversi ambienti
- Come per tutti i software di successo, le necessità degli utenti cambiano e si rendono necessarie nuove funzionalità
- Immaginiamo che, un bel giorno, gli utenti chiedano la possibilità di manipolare quadrati oltre che rettangoli



- L'ereditarietà è una relazione IsA
- In altre parole, perché un nuovo tipo di oggetto verifichi la relazione IsA con un tipo di oggetto esistente, la classe del nuovo oggetto deve essere derivata dalla classe dell'oggetto esistente
- Chiaramente, un quadrato è un rettangolo per tutti gli utilizzi e intenti normali
- Poiché vale la relazione IsA, è logico medellare Square come sottoclassse di Rectangle
- Questo utilizzo della relazione IsA è considerato da molti come una delle tecniche più importanti dell'Analisi Object Oriented
- Un quadrato è un tipo particolare di rettangolo, quindi la classe Square deve venire derivata dalla classe Rectangle

- Questo modo di pensare, però, può portare a problemi sottili, ma significativi
- In genere, tali problemi non vengono scoperti se non nella fase di implementazione dell'applicazione

```
public class Square : Rectangle
{
    public new double Width
    {
        get { return base.Width; }
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
    public new double Height
    {
        get { return base.Height; }
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

- È necessario ridefinire le proprietà `Width` e `Height` ...
- Notevoli differenze tra Java e C++/C#
 - ▶ In Java tutti i metodi sono virtuali
 - a parte i metodi `final`
 - ▶ In C++ / C# è possibile definire
 - sia metodi virtuali,
 - sia **metodi non virtuali** (non polimorfici)

```
...
Square s = new Square();
s.Width = 5; // 5 x 5
...
... Method1(s); //?
...
```

```
void Method1(Rectangle r)
{
    r.Width = 10;
}
```

- Se invochiamo `Method1` con un riferimento a un oggetto `Square`, l'oggetto `Square` non funzionerà correttamente in quanto l'altezza non verrà modificata
- Questa è una chiara violazione di LSP
- `Method1` non funziona per sottotipi dei suoi parametri
- Il motivo di questo malfunzionamento è che `Width` e `Height` non sono state dichiarate virtuali in `Rectangle`

```
public class Rectangle
{
```

```

    ...
    public virtual double Width
    { ... }
    public virtual double Height
    { ... }
    ...
}

public class Square : Rectangle
{
    public override double Width
    { ... }
    public override double Height
    { ... }
}

```

- Possiamo risolvere facilmente
- In ogni modo, quando la creazione di una classe derivata ci obbliga a modificare la classe base, spesso significa che il design è difettoso
- Infatti, viola l'OCP
- Potremmo rispondere argomentando che il vero difetto di progettazione è stato dimenticare di rendere virtuali Width e Height , e solo ora lo stiamo risolvendo
- Tuttavia, questo è difficile da giustificare poiché impostare l'altezza e la larghezza di un rettangolo sono operazioni estremamente primitive -con quale ragionamento le avremmo dovute rendere virtuali se non prevedendo l'esistenza del quadrato?
- A questo punto abbiamo due classi, Square e Rectangle, che apparentemente funzionano correttamente
- Indipendentemente da ciò che facciamo con un oggetto Square, questo rimarrà coerente con un quadrato matematico
- E indipendentemente da ciò che facciamo con un oggetto Rectangle, questo rimarrà un rettangolo matematico
- Inoltre, possiamo passare uno Square a una funzione che accetta un riferimento a un Rectangle e lo Square agirà comunque come un quadrato e rimarrà consistente
- Pertanto, potremmo concludere che il modello ora è consistente e corretto in sé
- Tuttavia, **un modello consistente in sé non è necessariamente consistente con tutti i suoi utenti!**

```

public void Scale(Rectangle rectangle)
{
    rectangle.Width = rectangle.Width * ScalingFactor;
    rectangle.Height = rectangle.Height * ScalingFactor;
}

```

- Scale invoca membri di ciò che crede essere un Rectangle
- Sostituendovi uno Square otterremo che il quadrato verrà ridimensionato due volte!

- E allora qui sta il **vero problema**: il programmatore che ha scritto Scale era giustificato nel presumere che la modifica della larghezza di un Rectangle lasci invariata la sua altezza?
- Chiaramente, il programmatore di Scale ha fatto questa ipotesi assai ragionevole
- Passare uno Square a funzioni i cui programmatori hanno fatto questa ipotesi provocherà problemi
- Pertanto, esistono funzioni che accettano riferimenti a oggetti Rectangle , ma non possono operare correttamente su oggetti Square
- Queste funzioni espongono una violazione di LSP
- L'aggiunta del sottotipo Square di Rectangle ha guastato queste funzioni: l'OCP è stato violato
- Cosa non va nel modello di Square e Rectangle?
 - Dopo tutto, un quadrato non è un rettangolo?
 - La relazione IsA non vale?
- No! Un quadrato sarà anche un rettangolo, ma un oggetto Square non è sicuramente un oggetto Rectangle
- Perché? Perché il **comportamento di un oggetto Square non è consistente con il comportamento di un oggetto Rectangle**
- Dal punto di vista comportamentale, uno Square non è un Rectangle!

E il software si basa proprio sul comportamento
- LSP chiarisce che **in OOD la relazione IsA riguarda il comportamento**
 - Non comportamento privato intrinseco, ma **comportamento pubblico estrinseco**
 - Comportamento da cui dipendono i clienti
- Ad esempio, l'autore di Scale dipendeva dal fatto che i rettangoli si comportano in modo tale che **le loro altezza e larghezza possano variare indipendentemente l'una dall'altra**
- Tale indipendenza delle due variabili è un **comportamento pubblico estrinseco** da cui probabilmente dipenderanno altri programmatori
- Affinché LSP possa valere, e con esso OCP, **tutti i sottotipi devono essere conformi al comportamento che i clienti si aspettano dalle classi base che utilizzano**
- La regola per le pre-condizioni e le post-condizioni per i sottotipi è:
“quando si ridefinisce una routine, si può sostituire **la sua pre-condizione solo con una più debole** e **la sua post-condizione solo con una più forte**”
- In altre parole, quando si utilizza un oggetto attraverso l'interfaccia della sua classe base, **L'utente conosce solo le pre-condizioni e le post-condizioni della classe base**
- Pertanto, le classi derivate non devono aspettarsi che tali utenti obbediscano a pre-condizioni più forti di quelle richieste dalla classe base
 - devono accettare tutto ciò che la classe base può accettare

- Inoltre, le classi derivate devono essere conformi a tutte le post-condizioni della classe base
 - i loro comportamenti e output non devono violare nessuno dei vincoli stabiliti per la classe base
- Il contratto di Rectangle
 - Altezza e larghezza sono indipendenti, si può modificarne una mantenendo costante l'altra
- Square viola il contratto della classe base

4.8 Il Quadrato è un Rettangolo?

- Guardando al codice di test della classe Rectangle possiamo avere qualche idea del contratto di Rectangle:

```
[TestFixture]
public class RectangleFixture
{
    [Test]
    public void SetHeightAndWidth()
    {
        Rectangle rectangle = new Rectangle();
        int expectedWidth = 3, expectedHeight = 7;
        rectangle.Width = expectedWidth;
        rectangle.Height = expectedHeight;
        Assertion.AssertEquals(expectedWidth, rectangle.Width);
        Assertion.AssertEquals(expectedHeight, rectangle.Height);
    }
}

[TestFixture] public class RectangleFixture
{
    [Test] public void SetHeightAndWidth()
    {
        Rectangle rectangle = GetShape();
        ...
    }
    protected virtual Rectangle GetShape()
    { return new Rectangle(); }
}

[TestFixture] public class SquareFixture : RectangleFixture
{
    protected override Rectangle GetShape()
    { return new Square(); }
}
```

4.9 Principi di Architettura dei Package

- **Reuse/Release Equivalency Principle** (REP)
- **Common Closure Principle** (CCP)
- **Common Reuse Principle** (CRP)

4.9.1 Release/Reuse Equivalency Principle

- *The granule of reuse is the granule of release*

- Un elemento riutilizzabile, sia esso un componente, una classe o un insieme di classi, non può essere riutilizzato a meno che non sia gestito da un sistema di rilascio di qualche tipo
- I clienti dovrebbero rifiutare di riutilizzare un elemento a meno che l'autore non prometta di tenere traccia dei numeri di versione e di mantenere le vecchie versioni per qualche tempo
 - un criterio per raggruppare le classi in package è il riutilizzo
- Poiché i pacchetti sono l'unità di rilascio in Java, sono anche l'unità di riutilizzo
- Pertanto, gli architetti farebbero bene a raggruppare in package le classi riutilizzabili assieme

4.9.2 Common Closure Principle

- *Classes that change together, belong together*
- Il lavoro per gestire, testare e rilasciare un pacchetto in un sistema di grandi dimensioni non è banale
- Più pacchetti cambiano in un dato rilascio, maggiore è il lavoro per ricostruire, testare e distribuire il rilascio
 - vorremmo **ridurre al minimo il numero di pacchetti che vengono modificati in un dato ciclo di rilascio del prodotto**
- Per raggiungere questo obiettivo, raggruppiamo assieme classi che pensiamo cambieranno insieme

4.9.3 Common Reuse Principle

- *Classes that aren't reused together should not be grouped together*
- Una dipendenza da un package è una dipendenza da tutto ciò che è contenuto nel package
- Quando un package cambia e il suo numero di rilascio viene aggiornato, tutti i client di quel package devono verificare di funzionare con il nuovo package - anche se nulla di ciò che hanno usato all'interno del package è effettivamente cambiato
- Pertanto, le classi che non vengono riutilizzate insieme non dovrebbero essere raggruppate insieme

4.9.4 Discussione

- Questi tre principi non possono essere soddisfatti contemporaneamente
- REP e CRP semplificano la vita ai riutilizzatori, mentre CCP semplifica la vita ai manutentori
- CCP cerca di rendere i package più grandi possibile (dopotutto, se tutte le classi stanno in un solo package, allora solo quel package cambierà)
- CRP, tuttavia, cerca di creare package molto piccoli
- All'inizio di un progetto, gli architetti possono impostare la struttura dei package in modo tale che CCP domini per facilità di sviluppo e manutenzione
- Successivamente, quando l'architettura si stabilizza, gli architetti possono rifattorizzare la struttura dei package per massimizzare REP e CRP per i riutilizzatori esterni

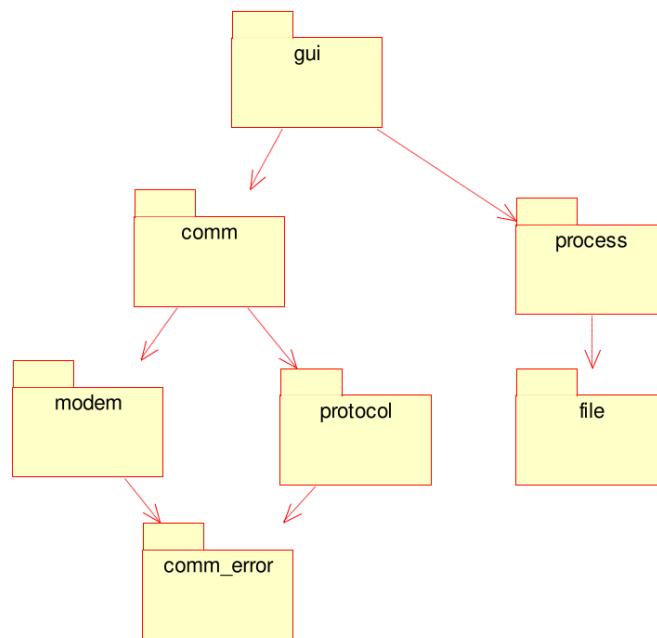
4.10 Relazioni tra i Package

- **Acyclic Dependencies Principle** (ADP)
- **Stable Dependencies Principle** (SDP)
- **Stable Abstractions Principle** (SAP)

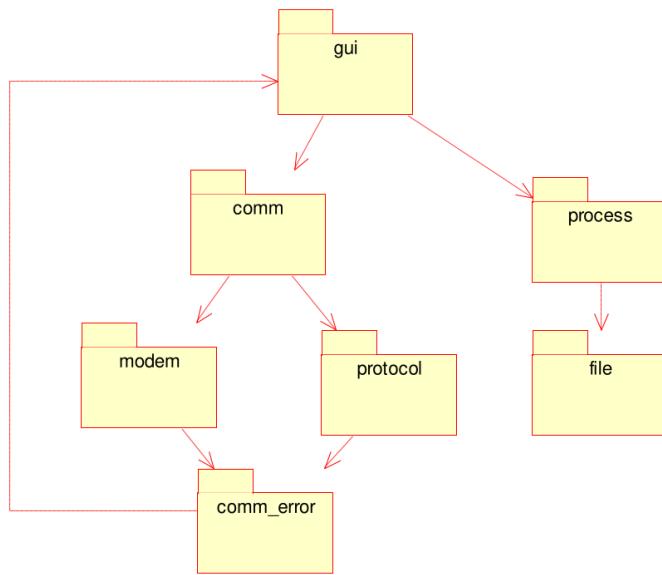
4.10.1 Acyclic Dependencies Principle

- *The dependencies between packages must not form cycles*
- Una volta apportate le modifiche a un package, gli sviluppatori possono rilasciare i package al resto del progetto
 - Prima di poter eseguire questo rilascio, tuttavia, devono verificare che il package funzioni
 - Per farlo, devono compilarlo e collegarlo a tutti i package da cui dipende
- Una singola dipendenza ciclica che sfugge al controllo può rendere l'elenco delle dipendenze molto lungo
- Quindi, qualcuno deve osservare la struttura delle dipendenze dei package con regolarità e interrompere i cicli ovunque compaiano

4.10.1.1 Esempio: Grafo dei Package Aciclico



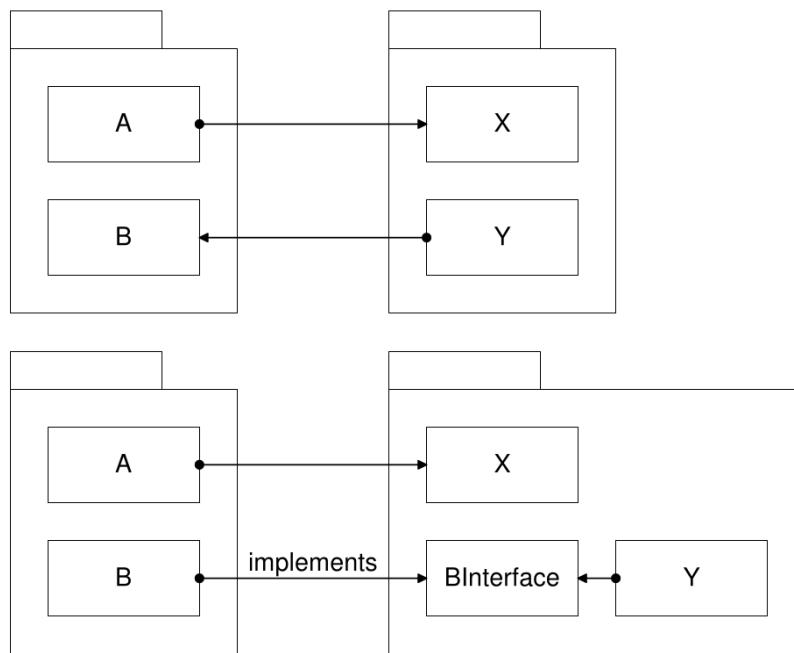
4.10.1.2 Esempio: Grafo dei Package Ciclico



4.10.1.3 Discussione

- Nello scenario aciclico per rilasciare il package protocol, gli ingegneri dovrebbero compilarlo con l'ultima versione del package comm_error ed eseguire i loro test
- Nello scenario ciclico per rilasciare il package protocol, gli ingegneri dovrebbero compilarlo con l'ultima versione di comm_error, gui, comm, process, modem, file ed eseguire i loro test
- Come rompere un ciclo:
 - Inframezzare un nuovo package
 - Aggiungere una nuova interfaccia

4.10.1.4 Rompere il Ciclo Introducendo un'Interfaccia

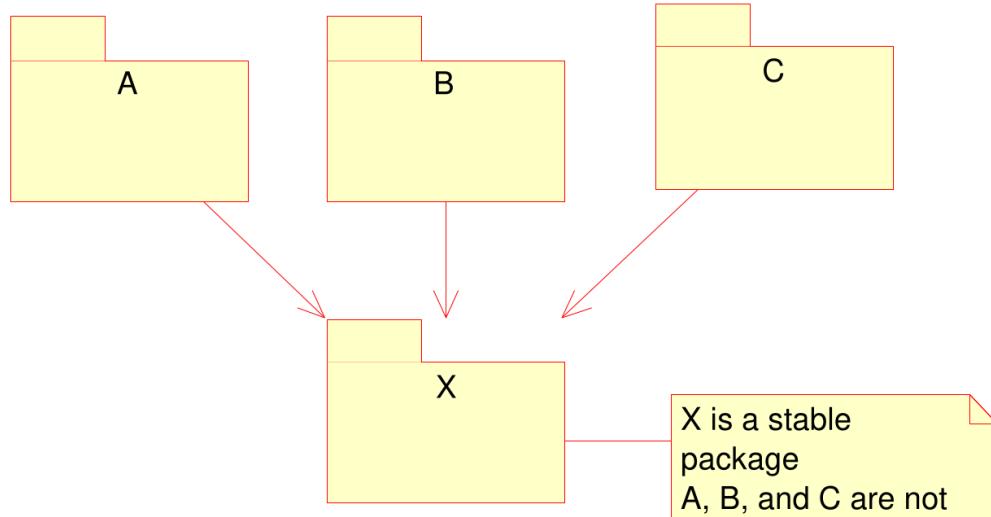


4.10.2 Stable Dependencies Principle

- *The dependencies between packages in a design should be in the direction of the stability of the packages.*
A package should only depend upon packages that are more stable than it is.
- I design non possono essere completamente statici

- Una certa volatilità è necessaria se il progetto deve essere mantenuto
- Raggiungiamo questo obiettivo conformandoci al CCP
- Alcuni package sono progettati per essere volatili, ci aspettiamo che cambino
- Un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare qualsiasi modifica con tutti i pacchetti dipendenti

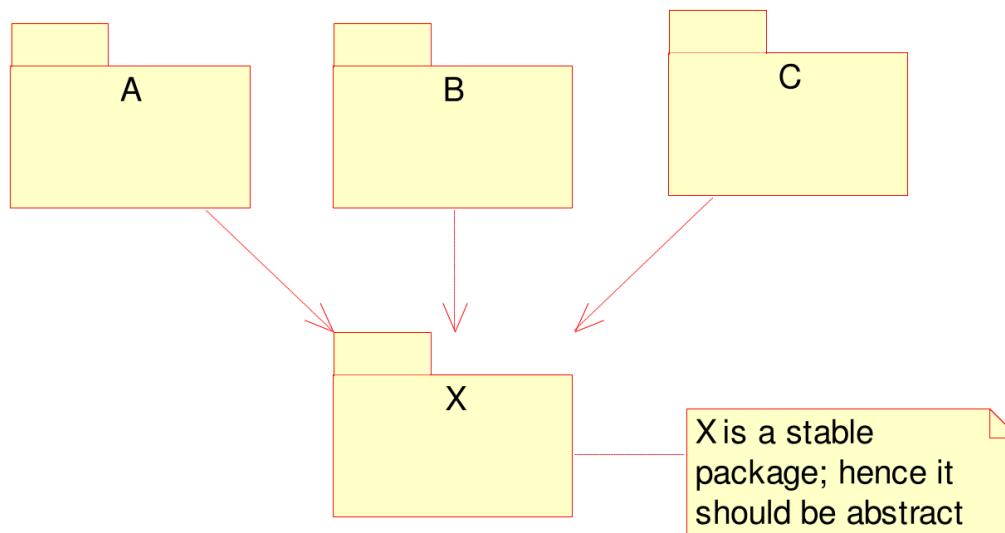
4.10.2.1 Esempio



4.10.3 Stable Abstractions Principle

- *Stable packages should be abstract packages*
- La stabilità è relativa alla quantità di lavoro richiesta per apportare una modifica
- Un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare le modifiche con tutti i pacchetti dipendenti

4.10.3.1 Esempio



4.10.4 Stable Dependencies/Abstractions Principles

4.10.4.1 Discussione

- I package in alto sono instabili e flessibili

- I package in basso sono molto difficili da modificare, perché ogni modifica a un package ne provoca altre
- I package altamente stabili nella parte inferiore del grafo delle dipendenze possono essere molto difficili da modificare, ma secondo l'OCP non devono essere difficili da estendere
- Se anche i pacchetti stabili in fondo sono molto astratti, possono essere facilmente estesi
- È possibile creare la nostra applicazione a partire da package instabili facili da modificare e package stabili facili da estendere

5 Sicurezza

5.1 Progettazione per la Sicurezza

- **La sicurezza non è qualcosa che può essere aggiunto al sistema**
- La sicurezza **deve essere progettata insieme al sistema** prima dell'implementazione
- “Sicurezza” **è anche un problema implementativo**: spesso le vulnerabilità sono introdotte durante la fase di implementazione
 - È possibile ottenere un’implementazione non sicura da una progettazione sicura
 - Non è possibile ottenere un’implementazione sicura partendo da una progettazione non sicura

5.2 Progettazione Architetturale

- La scelta dell’architettura del sistema influenza profondamente la sicurezza
- Un’architettura inappropriata non garantisce
 - riservatezza ed integrità delle informazioni
 - il livello di disponibilità richiesto
- Due problemi fondamentali vanno considerati quando si progetta l’architettura del sistema:
 - *Protezione*: come dovrebbe essere organizzato il sistema in modo che i beni critici possano essere protetti dagli attacchi esterni?
 - *Distribuzione*: come dovrebbero essere distribuiti i beni in modo da minimizzare gli effetti di un attacco andato a buon fine?
- I due problemi sono potenzialmente in conflitto:
 - se si mettono tutti i beni in un unico posto si può costruire un buon livello di protezione a un costo non eccessivo
 - se però la protezione fallisce tutti i beni sono compromessi
 - distribuire i beni porta ad un maggiore costo per la protezione
 - ci sono più possibilità che la protezione possa fallire se i beni sono distribuiti, ma se questo avviene vi è la perdita solo parziale dei beni
- Tipicamente la migliore architettura per fornire un alto grado di protezione è quella a layer
- I beni critici da proteggere sono posizionati al livello più basso
- Il numero di layer necessari varia da applicazione ad applicazione e dipende dalla criticità dei beni che devono essere protetti

- Per migliorare la protezione inoltre sarebbe bene che le credenziali di accesso ai diversi livelli fossero diverse tra loro
 - Esempio: se si adotta un meccanismo di accesso con password, ogni livello deve avere una propria password diversa da quelle degli altri livelli

5.2.1 Esempio

Protezione a livello di piattaforma

Autenticazione
di sistema

Autorizzazione
di sistema

Gestione
integrità file

Protezione a livello di applicazione

Login al
database

Autorizzazione
di database

Gestione della
transazione

Ripristino
del database

Protezione a livello di record

Autorizzazione
accesso record

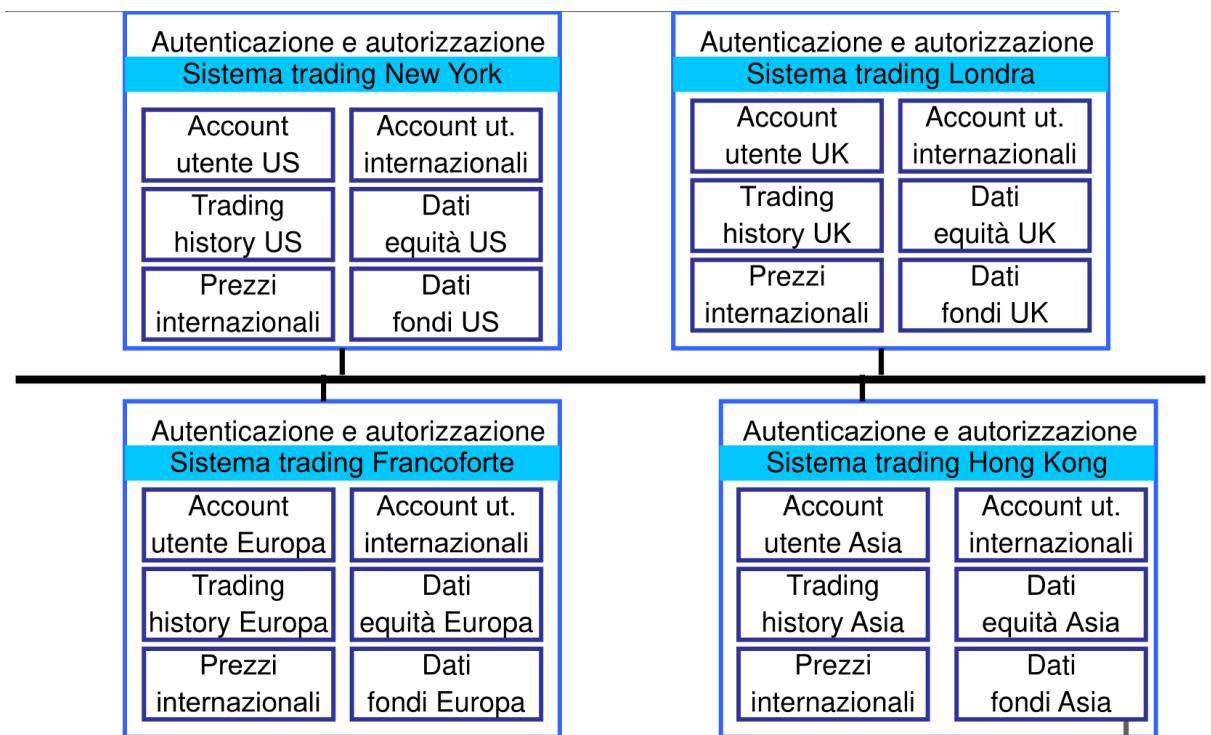
Cifratura
record

Gestione
integrità record

Dati dei pazienti

- Se la protezione dei dati è un requisito critico si potrebbe anche usare un'architettura client-server con i meccanismi di protezione nella macchina server
- La versione tradizionale client-server ha molte limitazioni
- Se la sicurezza viene compromessa
 - le perdite associate ad un attacco saranno alte Esempio: tutte le credenziali di accesso verranno compromesse
 - i costi di recupero saranno anch'essi elevati Esempio: tutte le credenziali di accesso al sistema andranno rigenerate
- Il sistema è inoltre maggiormente soggetto ad attacchi DoS che sovraccaricano il server
- Una possibile soluzione può essere quella di adottare una architettura distribuita in cui il server viene replicato in punti diversi della rete

5.2.2 Esempio



5.2.3 Esempio

- I beni del sistema sono distribuiti in diversi nodi della rete ognuno dei quali ha un proprio meccanismo di protezione dei dati
- I dati “più importanti” sono replicati nei diversi nodi
- Attacco ad uno specifico nodo:
 - alcuni beni non saranno disponibili
 - il sistema comunque potrà ancora funzionare e fornire i servizi più importanti
 - grazie alla replicazione, il ripristino dei dati nel nodo attaccato sarà più facile e meno costoso
- Tipico problema: lo stile architetturale più appropriato per la sicurezza potrebbe essere in conflitto con gli altri requisiti dell'applicazione (analisi trade-off)
- Esempio:
 - a. riservatezza dei dati memorizzati su un vasto database b. accesso molto veloce ai dati
 - Soddisfare entrambi i requisiti nella stessa architettura presenta molti problemi
 - a.
 - layer per garantire la riservatezza
 - diminuzione netta della velocità di accesso ai dati
 - b.
 - architettura “snella”
 - diminuzione netta della riservatezza
 - Tipico problema: lo stile architetturale più appropriato per la sicurezza potrebbe essere in conflitto con gli altri requisiti dell'applicazione (analisi trade-off)
 - Va valutato attentamente quale «requisito» è prioritario e l'architettura sarà scelta di conseguenza

5.3 Linee Guida di Progettazione

- Non ci sono regole rigide per ottenere un sistema sicuro

- **Differenti tipi di sistema richiedono differenti misure tecniche per ottenere un livello di sicurezza accettabile**
 - La posizione e i requisiti di diversi gruppi di utenti influenzano pesantemente **cosa è e cosa non è** accettabile
 - Ci sono comunque linee guida generali di ampia applicabilità per la progettazione di sistemi sicuri che possono fungere da:
 - mezzo per migliorare la consapevolezza dei problemi di sicurezza in un team di progettisti software
 - base per una lista di controlli da fare durante il processo di validazione del sistema
1. Basare le decisioni della sicurezza su una politica esplicita
 2. Evitare un singolo punto di fallimento
 3. Fallire in modo certo
 4. Bilanciare sicurezza e usabilità
 5. Essere consapevoli dell'esistenza dell'ingegneria sociale
 6. Usare ridondanza e diversità riduce i rischi
 7. Validare tutti gli input
 8. Dividere in compartimenti i beni
 9. Progettare per il deployment
 10. Progettare per il ripristino

5.3.1 Basare la Sicurezza su Policy

- La **Security Policy** è un documento di alto livello che definisce “cosa” è la sicurezza ma non “come” ottenerla
- La policy non dovrebbe definire i meccanismi usati per fornire e far rispettare la sicurezza
- Gli aspetti della security policy dovrebbero originare dai requisiti di sistema
- In pratica ciò è poco probabile, specie se viene adottato un processo di sviluppo rapido
- I progettisti dovrebbero quindi consultare la policy sia nelle decisioni di progettazione che nella loro valutazione

5.3.1.1 Progettazione delle Politiche

- Le politiche di sicurezza devono essere incorporate nella progettazione al fine di:
 - specificare come le informazioni possono essere accedute
 - quali precondizioni devono essere testate per l'accesso
 - a chi concedere l'accesso
- Tipicamente le politiche vengono rappresentate come un insieme di regole e condizioni
- Tali regole devono essere incorporate in uno specifico componente del sistema chiamato “**Security Authority**” che avrà il compito di far rispettare le politiche all'interno dell'applicazione
- A livello progettuale le politiche di sicurezza sono suddivise in sei specifiche categorie:
 - **Identity policies:** definiscono le regole per la verifica delle credenziali degli utenti

- ▶ **Access control policies:** definiscono le regole da applicare sia alle richieste di accesso alle risorse sia all'esecuzione di specifiche operazioni messe a disposizione dall'applicazione
- ▶ **Content-specific policies:** definiscono le regole da applicare a specifiche informazioni durante la memorizzazione e la comunicazione
- ▶ **Network and infrastructure policies:** definiscono le regole per controllare il flusso dei dati e il deployment sia delle reti che dei servizi infrastrutturali di hosting pubblici e privati
- ▶ **Regulatory policies:** definiscono le regole a cui l'applicazione deve sottostare per soddisfare i requisiti legali e di regolamentazione delle leggi in vigore nel Paese/Stato in cui il sistema opera
- ▶ **Advisor and information policies:** queste regole non sono imposte, ma sono caldamente consigliate in riferimento alle regole dell'organizzazione e al ruolo delle attività di business. Per esempio queste regole possono essere applicate per informare il personale sull'accesso ai dati sensibili o per stabilire comunicazioni commerciali con partner esterni

5.3.2 Evitare Punto Singolo di Fallimento

- Nei sistemi critici è buona norma di progettazione quella di cercare di **evitare un singolo punto di fallimento**
- Questo perché un singolo fallimento in una parte del sistema non si trasformi nel fallimento di tutto il sistema
- Per quanto riguarda la sicurezza questo significa che **non ci si dovrebbe affidare a un singolo meccanismo per assicurarla**, ma si dovrebbero impiegare differenti tecniche
- Questo viene spesso chiamato “**difesa in profondità**”
- Esempio: se si usa la password per autenticare si dovrebbe anche includere un meccanismo sfida e risposta

5.3.3 Fallire in Modo Certo

- Qualche tipo di fallimento è inevitabile in tutti i sistemi, ma i sistemi critici per la sicurezza dovrebbero sempre **fallire in modo sicuro**
- Non si dovrebbero avere procedure di fall-back meno sicure del sistema stesso
- Anche se il sistema fallisce **non deve essere consentito a un attaccante di accedere ai dati riservati**
- Esempio: i dati dei pazienti dovrebbero essere scaricati sul client all'inizio di ogni sessione clinica, così se il server fallisce i dati sono comunque mantenuti sul client. I dati vengono cifrati per non essere letti da personale non autorizzato. Questa pratica non vale per username e password, non è necessario inserire la password nel modello del dominio.

5.3.4 Bilanciare Sicurezza e Usabilità

- Sicurezza e usabilità sono spesso in contrasto
 - ▶ **per avere sicurezza bisogna introdurre un numero di controlli** che garantiscano che gli utenti siano autorizzati a usare il sistema e che nello stesso tempo agiscano in accordo alle politiche di sicurezza
 - ▶ questo inevitabilmente ricade sull'utente che ha bisogno di più tempo per imparare ad utilizzare il sistema

- Ogni volta che si aggiunge una caratteristica di sicurezza al sistema questo inevitabilmente diventa meno usabile
- A volte può diventare contro produttivo introdurre nuove caratteristiche di sicurezza a spese dell'usabilità
 - Esempio: obbligare l'utente all'adozione di password forti

5.3.5 Essere Consapevoli dell'Ingegneria Sociale

- **Ingegneria sociale**: trovare modi per convincere con l'inganno utenti accreditati al sistema a rivelare informazioni riservate
- Questi approcci si avvantaggiano della “**volontà di aiutare**” delle persone e della loro fiducia nell'organizzazione
- Dal punto di vista della progettazione contrastare l'ingegneria sociale **è quasi impossibile**
- Se la sicurezza è molto critica non si dovrebbe affidarsi solo a meccanismi di autenticazione basati su password, ma bisognerebbe utilizzare meccanismi di autenticazione forte
- Meccanismi di log che tracciano sia la locazione che l'identità dell'utente e programmi di analisi del log potrebbero essere utili ad identificare brecce nella sicurezza

5.3.6 Usare Ridondanza e Diversità

- Ridondanza significa **mantenere più di una versione** del software e dei dati nel sistema
- Diversità significa **che le diverse versioni del sistema non dovrebbero usare la stessa piattaforma o essere basati sulle stesse tecnologie**
- In questo modo una vulnerabilità della piattaforma o della tecnologia non influirà su tutte le versioni e non condurrà a un comune punto di fallimento
- Esempio:
 - mantenere i dati dei pazienti sia sul client che sul server
 - client e server devono avere un diverso sistema operativo
 - attacco basato su vulnerabilità del SO non influenza sia client che server

5.3.7 Validare tutti gli Input

- Un comune attacco al sistema consiste nel fornire input inaspettati che causano un comportamento imprevisto
 - crash
 - perdita della disponibilità del servizio
 - esecuzione di codice malevolo
- Tipici esempi sono buffer overflow e SQL injection
- Si possono evitare molti di questi problemi progettando la validazione dell'input in tutto il sistema
- Nei requisiti dovrebbero essere definiti tutti i controlli che devono essere applicati
- Bisogna usare la conoscenza dell'input per definire questi controlli

5.3.8 Dividere in Compartimenti i Beni

- **Compartimentalizzare** significa organizzare le informazioni nel sistema in modo che gli **utenti abbiano accesso solo alle informazioni necessarie** piuttosto che a tutte le informazioni del sistema

- Gli effetti di un attacco in questo modo sono più contenuti: qualche informazione sarà persa o danneggiata, ma è poco probabile che tutte le informazioni del sistema siano coinvolte
- Esempio:
 - lo staff clinico può avere accesso soltanto ai record dei pazienti che hanno un appuntamento o sono ricoverati nella clinica
 - esiste un meccanismo per gestire accessi inaspettati

5.3.9 Progettazione per il Deployment

- Molti problemi di sicurezza sorgono perché il sistema **non viene configurato correttamente** al momento del deployment
- Bisogna sempre progettare il sistema in modo che
 - siano inclusi programmi di utilità per semplificare il deployment
 - verificare potenziali errori di configurazione e omissioni nel sistema di deployment

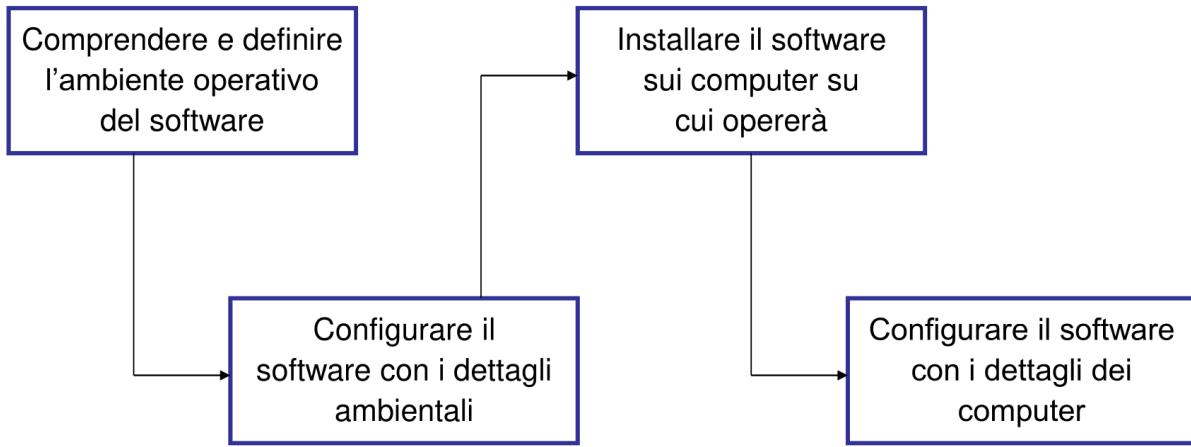
5.3.10 Progettazione per il Ripristino

- Bisogna sempre progettare il sistema con l'assunzione che gli errori di sicurezza possano accadere
- Si deve quindi pensare a come ripristinare il sistema dopo possibili errori e riportarlo a uno stato operazionale sicuro
- Esempio:
 - persone non autorizzate accedono ai dati dei pazienti
 - non è noto come abbiano ottenuto le credenziali per l'accesso
 - occorre quindi cambiare tutte le credenziali del sistema in modo che la persona non autorizzata non abbia accesso al meccanismo di cambiamento delle password

5.4 Progettazione per il Deployment

- Il deployment di un sistema coinvolge:
 - configurazione del sistema per operare nell'ambiente:
 - semplice impostazione di parametri delle preferenze degli utenti
 - definizione di regole e modelli di business che governano l'esecuzione del software
 - installazione del sistema sui computer dell'ambiente
 - configurazione del sistema installato

5.4.1 Deployment del Software



- Nella fase di deployment vengono spesso introdotte in modo accidentale delle vulnerabilità
- Esempio:
 - il software deve spesso essere configurato con una lista di utenti autorizzati
 - quando il software è rilasciato questa lista consiste di un login per l'amministratore generico come “admin” e la password di default è “password”
 - come prima azione l'amministratore dovrebbe modificare i dati di login, ma è molto facile dimenticare di farlo
 - un attaccante che conosce il login di default potrebbe essere capace di guadagnare privilegi di accesso al sistema
- La configurazione e il deployment sono spesso visti solo come problemi di amministrazione e quindi al di fuori del processo di ingegnerizzazione
- I progettisti software hanno la responsabilità di progettare per il deployment
- Bisogna sempre fornire supporti per il deployment che riducano la probabilità che gli amministratori compiano degli errori quando configurano il software
- Esistono delle linee guida per la progettazione per il deployment

5.4.2 Linee Guida

1. Includere supporto per visionare e analizzare le configurazioni
2. Minimizzare i privilegi di default
3. Localizzare le impostazioni di configurazione
4. Fornire modi per rimediare a vulnerabilità di sicurezza

5.4.2.1 Supporto per le Configurazioni

- Si devono sempre includere **programmi di utilità** che consentano agli amministratori di esaminare la configurazione corrente del sistema
- Sorprendentemente questi programmi **mancano** nella maggior parte dei sistemi software
- Gli utenti sono spesso frustrati dalla difficoltà di trovare i dettagli della configurazione
 - per un quadro completo della configurazione spesso occorre visionare diversi menu e questo porta a errori e omissioni

- Idealmente in fase di visualizzazione delle configurazioni si dovrebbero evidenziare impostazioni critiche per la sicurezza

5.4.2.2 Minimizzare i Privilegi di Default

- Il software deve essere progettato in modo tale che la **configurazione di default fornisca i minimi privilegi essenziali**
- In questo modo vengono limitati i danni di un possibile attacco
- Esempio:
 - l'autenticazione di default dell'amministratore dovrebbe solo consentire l'accesso a un modulo che permette all'amministratore di inserire nuove credenziali
 - non dovrebbe essere consentito l'accesso a nessuna altra funzionalità
 - quando vengono modificate le credenziali, quella di default dovrebbe essere automaticamente cancellata

5.4.2.3 Localizzare le Impostazioni di Configurazione

- Quando si progetta il supporto per le configurazioni del sistema bisognerebbe assicurarsi che ogni risorsa che appartiene alla stessa parte del sistema venga configurata nella stessa posizione
- Se le informazioni di configurazione non sono localizzate
 - è facile dimenticarsi di farlo
 - può capitare di non essere a conoscenza dell'esistenza di meccanismi per la sicurezza già inclusi nel sistema
 - se tali meccanismi presentano configurazioni di default si potrebbe essere esposti ad attacchi

5.4.2.4 Rimediare a Vulnerabilità

- Bisogna includere **meccanismi diretti** per:
 - aggiornare il sistema
 - riparare le vulnerabilità di sicurezza che vengono scoperte
- Questi potrebbero includere
 - verifiche automatiche per aggiornamenti di sicurezza
 - download di tali aggiornamenti non appena sono disponibili
- Va comunque considerato che gli aggiornamenti devono coinvolgere contemporaneamente centinaia di PC su cui tipicamente il software è installato

5.5 Testare la Sicurezza

- Il test di un sistema gioca **un ruolo chiave** nel processo di sviluppo software e dovrebbe essere eseguito con molta attenzione
- È quindi sorprendente che **l'area dei test della sicurezza sia quella più trascurata durante lo sviluppo del sistema**
- Questo può essere attribuito a diversi fattori:
 - mancanza di comprensione dell'importanza dei test relativi alla sicurezza
 - mancanza di tempo
 - mancanza di conoscenza su come svolgere un test di sicurezza
 - mancanza di tool integrati per compiere test
- Il test della sicurezza è un lavoro molto lungo e tedioso, spesso molto più complesso dei test funzionali che vengono svolti normalmente

- Inoltre esso coinvolge diverse discipline
 - ci sono tradizionali test per accertare la sicurezza dei requisiti applicativi che possono essere svolti normalmente dal team di testing
 - ma esistono dei test non funzionali di “rottura” del sistema che devono essere condotti da esperti di sicurezza
 - Black box testing
 - White box testing

5.5.1 Black Box Testing

- Questo test ha come assunzione di base la non conoscenza dell'applicazione
- I tester affrontano l'applicazione come farebbe un attaccante
 - indagando sulle informazioni riguardanti la struttura interna
 - successivamente applicano un insieme di tentativi di violazione del sistema basati sulle informazioni ottenute
- Esempio: se un URL di una applicazione contiene una estensione “.cgi” allora può essere inferito che l'applicazione è stata sviluppata con la tecnologia CGI e applicare quindi le ben conosciute tecniche di violazione di questa tecnologia
- I tester possono impiegare una varietà di strumenti per scansionare e indagare l'applicazione
 - ci sono centinaia di tool in rete per l'hacking di applicazioni che permettono di “scandagliare” le porte dei sistemi perpetuando attacchi sfruttando le debolezze ben conosciute di svariati linguaggi di programmazione
- Questo test non prende in esame solo debolezze del codice, ma vengono svolti test mirati anche al livello infrastrutturale
 - errori di configurazione di reti e host
 - fallo di sicurezza nelle macchine virtuali
 - problemi legati ai linguaggi di implementazione

5.5.2 White Box Testing

- Questo test ha come assunzione di base la completa conoscenza dell'applicazione
- I tester hanno accesso a tutte le informazioni di configurazione e anche al codice sorgente
- Essi operano una revisione del codice cercando possibili debolezze
- Inoltre scrivono test per stabilire come trarre vantaggio dalle debolezze scoperte
- Tipicamente questi tester sono ex-sviluppatori o persone che conoscono molto bene l'ambiente di sviluppo
- I tool a disposizione differiscono molto da quelli usati nel black box test
- Tipicamente sono tool di debugging che consentono di trovare bachi e vulnerabilità specifici del sistema
- I bachi tipici riguardano problemi di corsa critici e la mancanza di verifica dei parametri di input e sono specifici di ogni applicazione

- Questi test portano a scoprire anche altri problemi come i memory leak e problemi di prestazione che contribuiscono al danneggiamento della disponibilità e dell'affidabilità dell'intero sistema

5.6 Capacità di Sopravvivenza del Sistema

TUTTO QUESTO CAPITOLO NON È DA STUDIARE

- Con il termine **capacità di sopravvivenza** (*survivability*) si intende la capacità del sistema di continuare a fornire i servizi essenziali agli utenti legittimi
 - mentre è sotto attacco
 - dopo che parti del sistema sono state danneggiate come conseguenza di un attacco o di un fallimento
- La capacità di sopravvivenza è una proprietà dell'intero sistema, non dei singoli componenti di questo
- Il lavoro sulla capacità di sopravvivenza è molto critico poiché l'economia e la vita sociale dipendono da infrastrutture controllate da computer
- L'analisi e la progettazione della capacità di sopravvivenza dovrebbero essere parte del processo di ingegnerizzazione dei sistemi sicuri
- **La disponibilità dei servizi critici è l'essenza della sopravvivenza**
- Questo significa conoscere
 - quali sono i servizi **maggiormente critici**
 - come questi servizi possono essere compromessi
 - qual è la **qualità minima dei servizi** che deve essere mantenuta
 - come proteggere questi servizi
 - come **ripristinare velocemente** il sistema se i servizi diventano non disponibili

5.6.1 Esempio

- Un sistema informatico che gestisce l'invio delle ambulanze in risposta alle chiamate di emergenza
- Servizi:
 - prendere le chiamate e inviare le ambulanze
 - log delle chiamate
 - gestione locazione delle ambulanze
- Il servizio critico è quello legato a prendere le chiamate e inviare le ambulanze perché necessita di un processo real-time per la gestione degli eventi

5.6.2 Analisi della Sopravvivenza

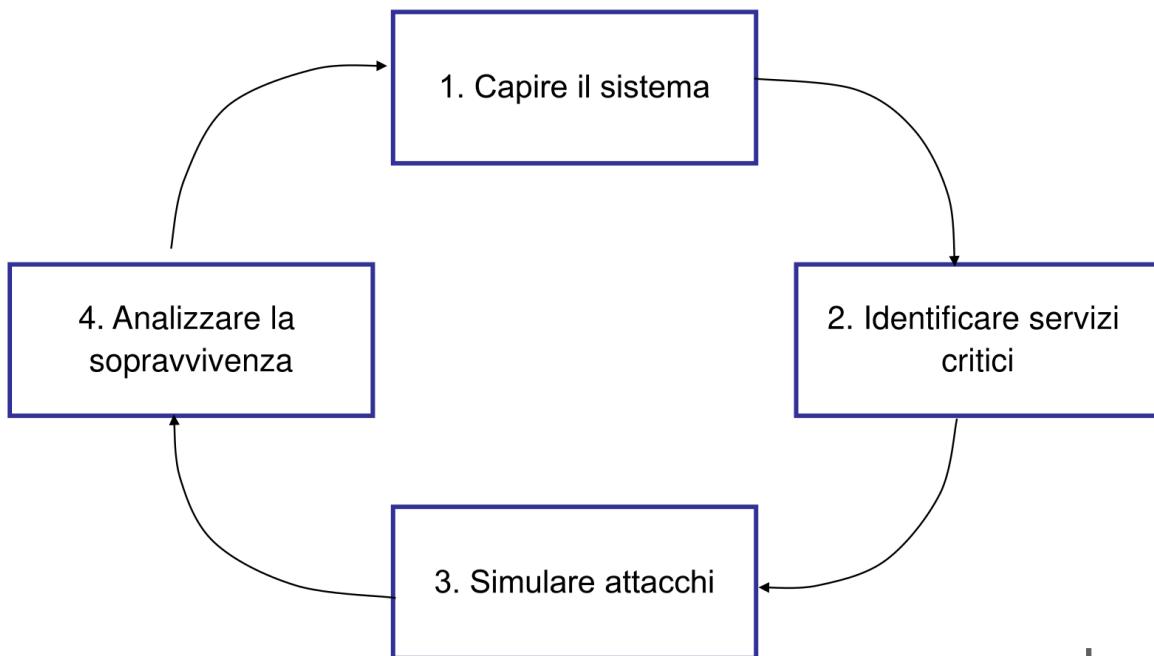
- Il Survivable Analysis Systems è un metodo di analisi ideato agli inizi del 2000 per:
 - valutare le vulnerabilità nel sistema
 - supportare la progettazione di architetture e caratteristiche che promuovono la sopravvivenza del sistema
- In questo metodo la sopravvivenza del sistema
 - dipende da tre strategie complementari
 - è un processo a 4 fasi

5.6.3 Strategie

- **Resistenza:**

- evitare problemi costruendo all'interno del sistema le capacità di respingere attacchi
- es: firma digitale per l'autenticazione
- **Identificazione:**
 - individuare problemi costruendo all'interno del sistema le capacità di riconoscere attacchi e fallimenti e valutare il danno risultante
 - es: aggiungere checksum ai dati critici
- **Ripristino:**
 - tollerare problemi costruendo all'interno del sistema le capacità di fornire servizi essenziali durante un attacco
 - ripristinare le complete funzionalità dopo l'attacco

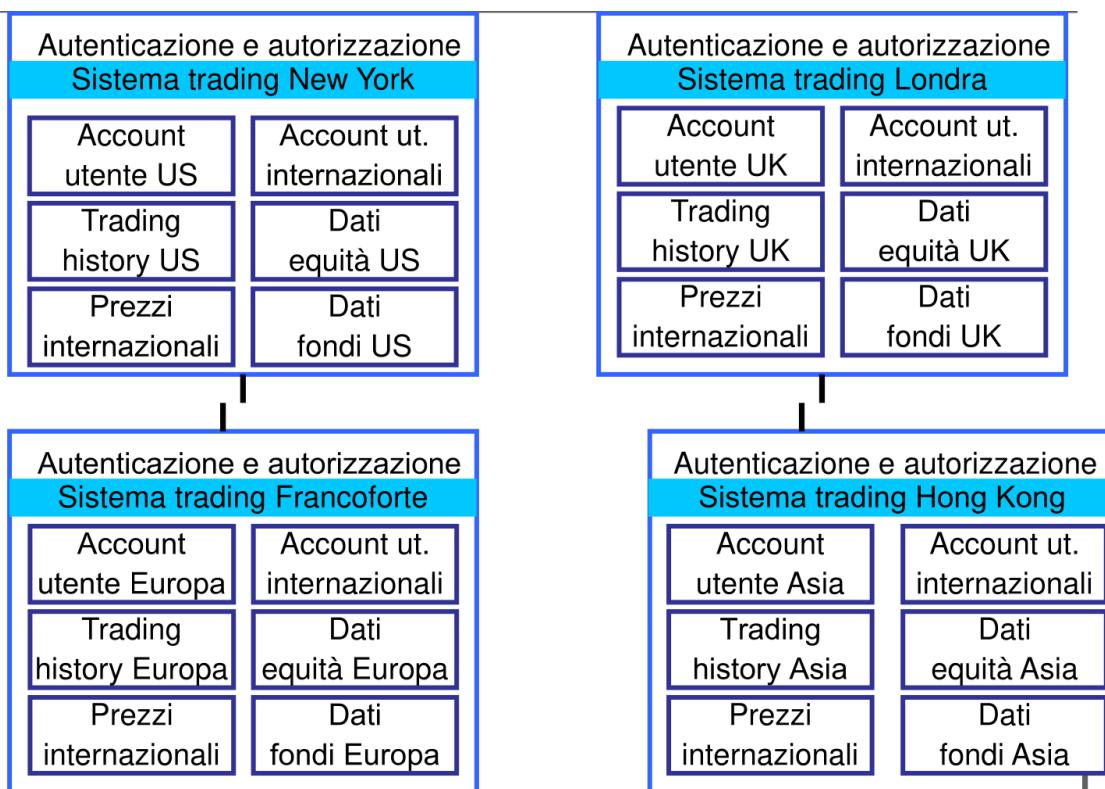
5.6.4 Fasi Analisi di Sopravvivenza



5.6.5 Principali Attività

- **Capire il sistema:** riesaminare gli obiettivi del sistema, i requisiti e l'architettura
- **Identificare servizi critici:** identificare i servizi che devono essere mantenuti e i componenti che devono svolgere tale compito
- **Simulare gli attacchi:** identificare gli scenari o i casi d'uso dei possibili attacchi insieme ai componenti influenzati da questi attacchi
- **Analizzare la sopravvivenza:** identificare
 - i componenti che sono sia essenziali che a rischio
 - le strategie di sopravvivenza basate su resistenza, identificazione e ripristino

5.6.6 Esempio



5.6.7 Esempio

- Sistema che gestisce l'equità dei prezzi nei mercati internazionali
- Qual è il minimo supporto che il sistema fornisce già per la sopravvivenza?
 - gli account dei clienti e i prezzi internazionali sono replicati in tutti i nodi
- Servizio chiave che deve sempre essere mantenuto: capacità di piazzare ordini
 - mantenere l'integrità dei dati
 - ordini accurati e che riflettano le vendite e gli acquisti degli utenti

5.6.8 Esempio

- Per mantenere questo servizio ci sono tre componenti del sistema
 - *autenticazione dell'utente*: consente agli utenti autorizzati di accedere al sistema
 - quotazione dei prezzi: consente che la vendita e l'acquisto degli stock dei beni siano quotati
 - piazzamento ordini: consente di vendere o comprare beni al dato prezzo di mercato
- Questi componenti fanno uso dei dati
 - relativi agli utenti
 - accedono al database delle transazioni

5.6.9 Esempio

- Possibili attacchi al sistema:
 - utente malevolo guadagna le credenziali di accesso di utente verso cui nutre forte rancore
 - vengono piazzati ordini di vendita e acquisto in modo tali da causare seri problemi all'utente legittimo
 - utente non autorizzato corrompe il database delle transazioni guadagnando permessi per eseguire direttamente query SQL
 - riconciliare gli acquisti e le vendite diventa quindi impossibile

5.6.10 Esempio

Attacco	Resistenza	Identificazione	Ripristino
Ordini malevoli	Usare una password diversa da quella di login per piazzare ordini	Mandare una copia degli ordini per e-mail all'utente autorizzato Mantenere storia degli ordini e controllare pattern inusuali di trading	Fornire meccanismi automatici di "undo" e ripristinare l'account Risarcire l'utente per le perdite subite Assicurarsi contro le perdite
Corruzione database	Gli utenti privilegiati necessitano di meccanismi di autenticazione forte	Mantenere una copia a sola lettura delle transazioni effettuate sui vari nodi su un server internazionale. Controllare periodicamente per individuare corruzioni Mantenere una checksum crittografata di tutte le transazioni	Ripristinare il database con le copie di backup Fornire un meccanismo di ripetizione degli scambi dopo un certo tempo per ricreare il database delle transazioni

5.6.11 Capacità di Sopravvivenza

- Aggiungere le tecniche di sopravvivenza costa soldi
- Spesso le aziende sono molto riluttanti ad investire sulla sopravvivenza, specie se non hanno mai subito attacchi e perdite
- È comunque sempre buona norma investire nella sopravvivenza prima piuttosto che dopo aver già subito un attacco
- L'analisi della sopravvivenza non è ancora inclusa nella maggior parte dei processi di ingegnerizzazione del software
- Con la crescita dei sistemi critici sembra probabile che questo tipo di analisi sarà sempre più utilizzato

5.7 Conclusioni

- La sicurezza deve **essere onnipresente attraverso tutto il ciclo di sviluppo del software**
- Inoltre la sicurezza deve essere tenuta in considerazione anche attraverso tutti gli strati dell'infrastruttura su cui l'applicazione viene sviluppata
- Per ottenere questo è imperativa l'adozione di un processo che tenga in considerazione le problematiche relative alla sicurezza sin dalle prime fasi dello sviluppo del sistema
- È inoltre necessario che vengano compiuti **severi test periodici** a verifica del livello di sicurezza del sistema

6 Dalla Progettazione all'Implementazione

6.1 Introduzione

- Arrivati a questo punto apparentemente occorre "solamente" implementare tutte le classi che abbiamo individuato nelle varie fasi (analisi, progettazione, ecc.)
- "Sfortunatamente" anche a questo livello occorre effettuare delle scelte progettuali che hanno un impatto sulle caratteristiche del SW (efficienza, riusabilità, ecc.)
- Il **progetto di dettaglio** rappresenta una descrizione del sistema molto vicina alla codifica, ovvero che la vincola in maniera sostanziale

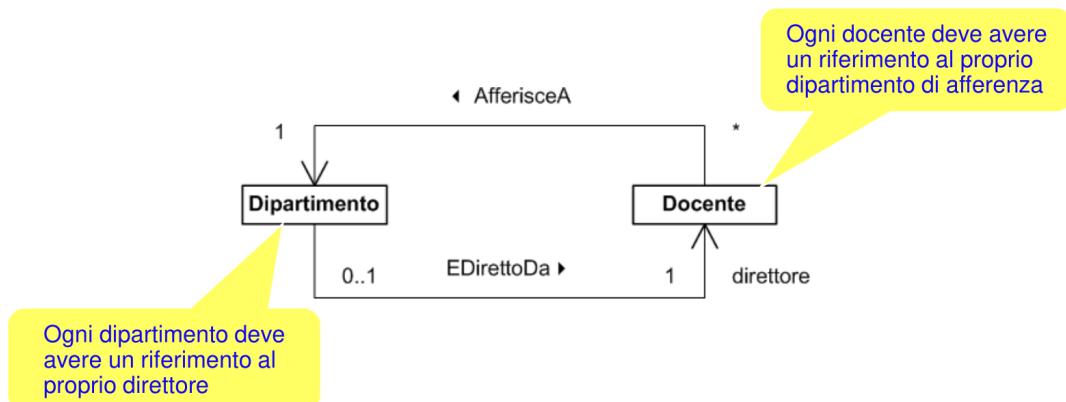
- Per esempio, descrivendo non solo le classi in astratto ma anche i loro attributi e metodi, con relativi tipi e firma

6.2 Progettazione di Dettaglio

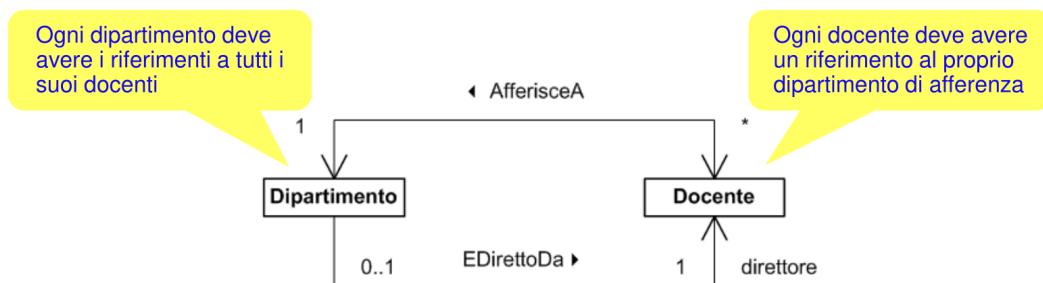
- Durante la progettazione di dettaglio è necessario definire
 - Tipi di dato che non sono stati definiti nel modello OOA
 - Navigabilità delle associazioni tra classi e relativa implementazione
 - Strutture dati necessarie per l'implementazione del sistema
 - Operazioni necessarie per l'implementazione del sistema
 - Algoritmi che implementano le operazioni
 - Visibilità di classi, (attributi,) operazioni, ...

6.2.1 Navigabilità di un'Associazione

- Possibilità di spostarsi da un qualsiasi oggetto della classe origine a uno o più oggetti della classe destinazione (a seconda della molteplicità)
- I messaggi possono essere inviati solo nella direzione della freccia Ogni docente deve avere un riferimento al proprio dipartimento di afferenza



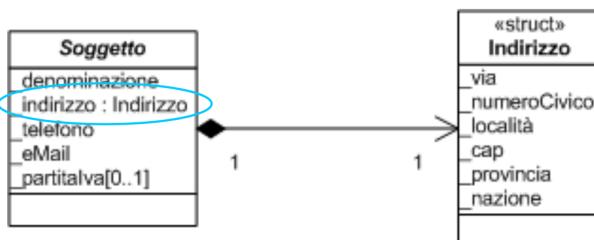
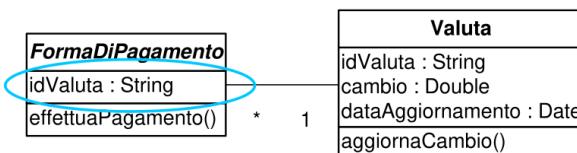
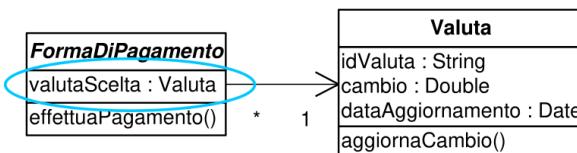
- A livello di analisi del problema**, le associazioni di composizione e di aggregazione hanno una direzione precisa detti **A il contenitore** e **B l'oggetto contenuto**, è A che contiene B, e non viceversa
- A livello implementativo**, un'associazione può essere
 - mono-direzionale** quando da A si deve poter accedere a B, ma non viceversa
 - bi-direzionale** quando da A si deve poter accedere a B e da B si deve poter accedere velocemente ad A
- Dal punto di vista implementativo, la **bi-direzionalità**
 - è molto efficiente
 - ma occorre tenere sotto controllo la consistenza delle strutture dati utilizzate per la sua implementazione



6.2.2 Implementazione delle Associazioni

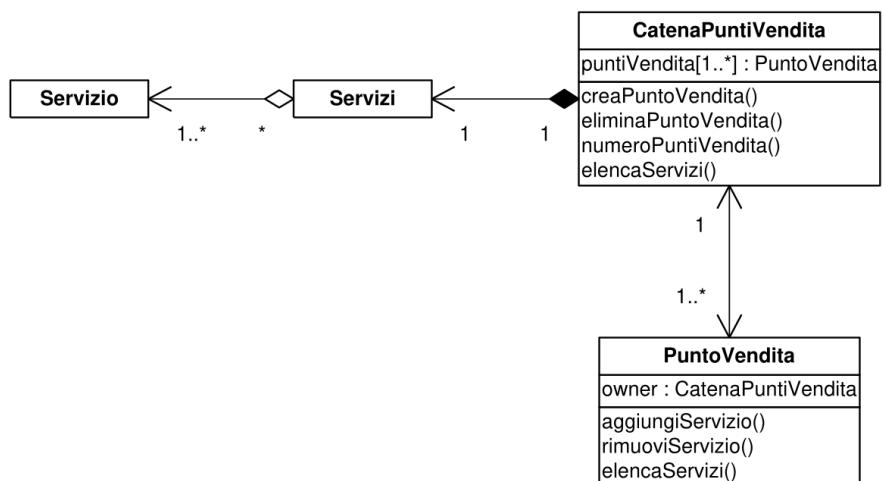
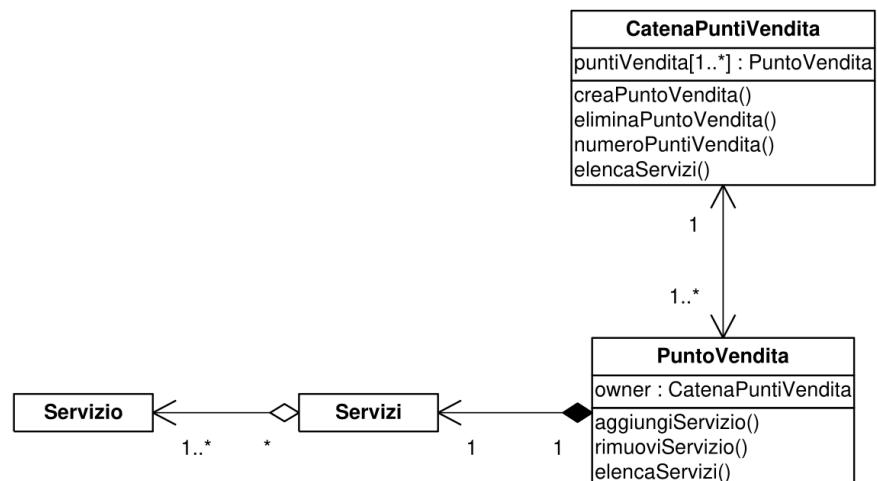
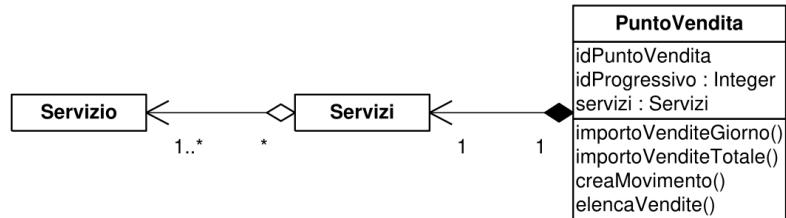
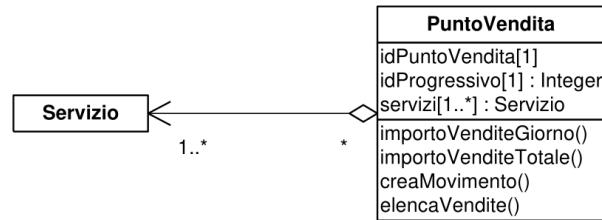
6.2.2.1 Associazioni con molteplicità 0..1 o 1..1

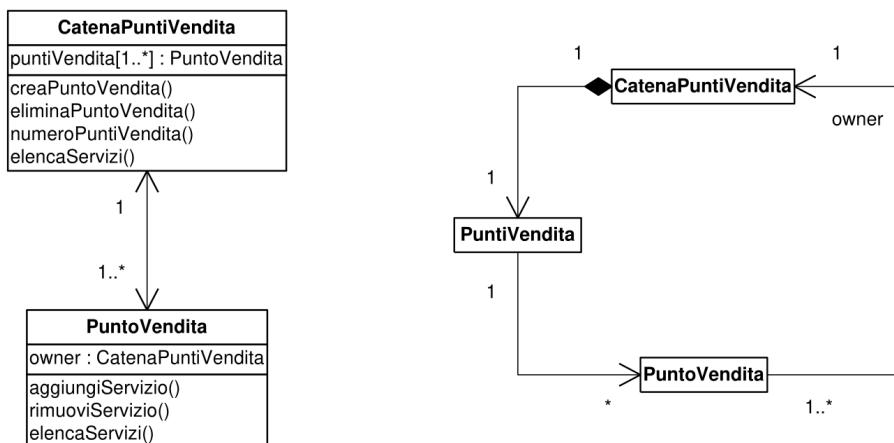
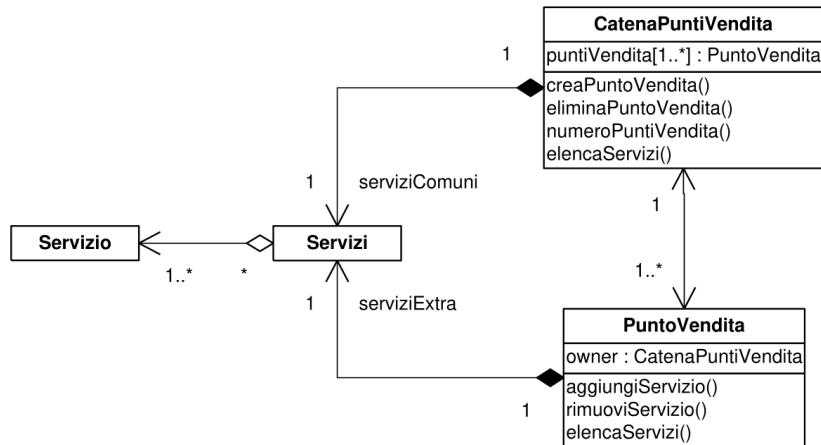
- Aggiungere alla classe cliente un attributo membro che rappresenta
 - il **riferimento all'oggetto** della classe fornitore
 - e/o l'**identificatore univoco** dell'oggetto della classe fornitore (solo se persistente)
 - o il **valore dell'oggetto** della classe fornitore (solo nel caso di **composizione** e **molteplicità 1..1**)



6.2.2.2 Associazioni con molteplicità 0..* o 1..*

- Aggiungere alla classe cliente un attributo membro che referenzia un'**istanza di una classe contenitore**
- Una **classe contenitore** è una classe le cui istanze sono **collezioni** di (riferimenti
 - a) oggetti della classe fornitore
- La classe contenitore può essere
 - realizzata, oppure
 - presa da una libreria (preferibilmente)





6.2.3 Classi Contenitore

- Una **classe contenitore** (o semplicemente contenitore) è una classe le cui istanze contengono oggetti di altre classi
- Se gli oggetti contenuti sono in **numero fisso**, è sufficiente un vettore predefinito del linguaggio
- Se gli oggetti contenuti sono in **numero variabile**, un vettore predefinito non basta e occorre una classe contenitore
- Esempi di classi contenitore sono
 - Vettori, stack, liste, alberi, ...
- Funzionalità minime di una classe contenitore
 - **Inserire, rimuovere, trovare** un oggetto in una collezione
 - **Enumerare** (iterare su) gli oggetti della collezione
- I contenitori possono essere classificati in funzione
 - del modo in cui contengono gli oggetti
 - **contenimento per riferimento**: gli oggetti sono reference type
 - **contenimento per valore**: gli oggetti sono value type
 - dell'omogeneità o eterogeneità di tali oggetti
 - **oggetti omogenei**: tutti gli oggetti contenuti sono dello stesso tipo
 - **oggetti eterogenei**: gli oggetti contenuti possono essere di tipo diverso

6.2.3.1 Contenimento per Riferimento

- L'oggetto contenuto **esiste per conto proprio**

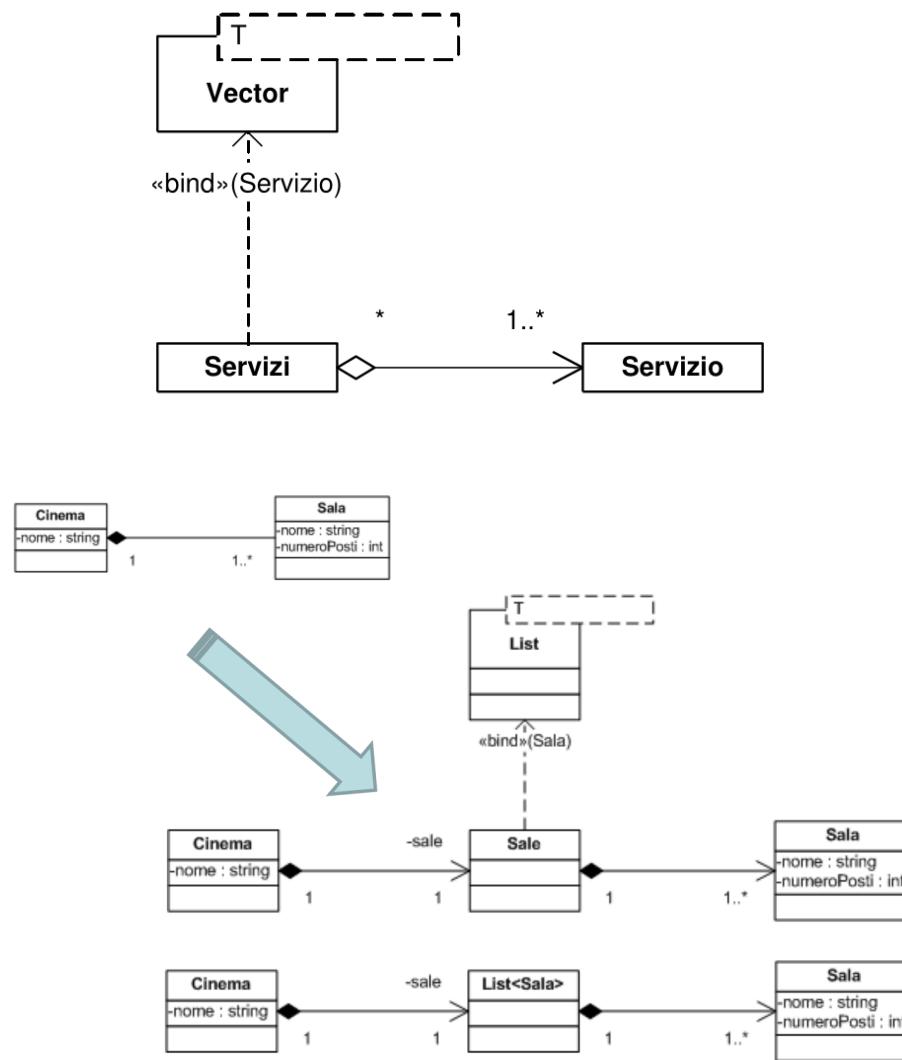
- L'oggetto contenuto può essere **in più contenitori contemporaneamente**
- Quando un oggetto viene inserito in un contenitore, **non viene duplicato** ma ne viene memorizzato solo il riferimento
- La distruzione del contenitore non comporta la **distruzione degli oggetti contenuti**

6.2.3.2 Contenimento per Valore

- L'oggetto contenuto
 - viene memorizzato nella struttura dati del contenitore
 - esiste solo in quanto contenuto fisicamente in un altro oggetto
- Quando un oggetto deve essere inserito in un contenitore, **viene duplicato**
- La distruzione del contenitore comporta la **distruzione degli oggetti contenuti**

6.2.3.3 Contenimento di Oggetti Omogenei

- Per implementare **contenitori di oggetti omogenei** (sia per valore, sia per riferimento) sono ideali le **classi generiche**
- Il tipo degli oggetti contenuti viene lasciato generico e ci si concentra sugli algoritmi di gestione della collezione di oggetti
- Quando serve una classe contenitore di oggetti appartenenti a una classe specifica, **è sufficiente istanziare la classe generica**, specificando il tipo desiderato



6.2.3.4 Contenimento di Oggetti Eterogenei

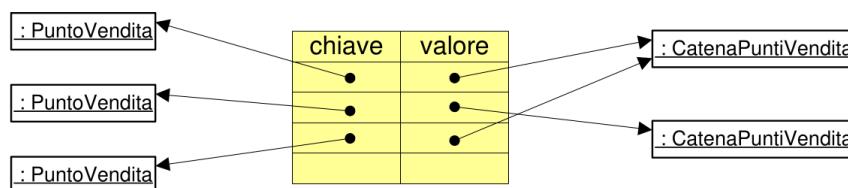
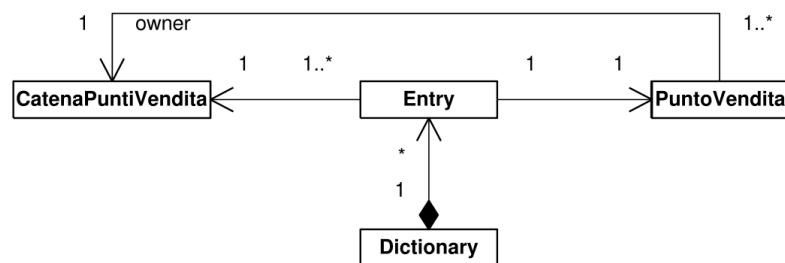
- Per implementare **contenitori di oggetti eterogenei** (solo per riferimento) è necessario usare l'**ereditarietà** e sfruttare la proprietà che un puntatore alla

superclasse radice della gerarchia può puntare a un’istanza di una qualunque sottoclasse

- La classe contenitore può essere generica, ma il tipo deve essere la **superclasse radice della gerarchia** (nel peggior dei casi, object)

6.3 Implementazione delle Associazioni

- Un modo alternativo per implementare un’associazione tra due oggetti è tramite un **dizionario**
- Un dizionario è un tipo particolare di contenitore, che associa due oggetti: **la chiave e il rispettivo valore**
- La chiave
 - Può essere un oggetto qualsiasi, non necessariamente una stringa o un intero
 - Deve essere unica
- Il dizionario, data una chiave, ritrova in modo efficiente il valore ad essa associato



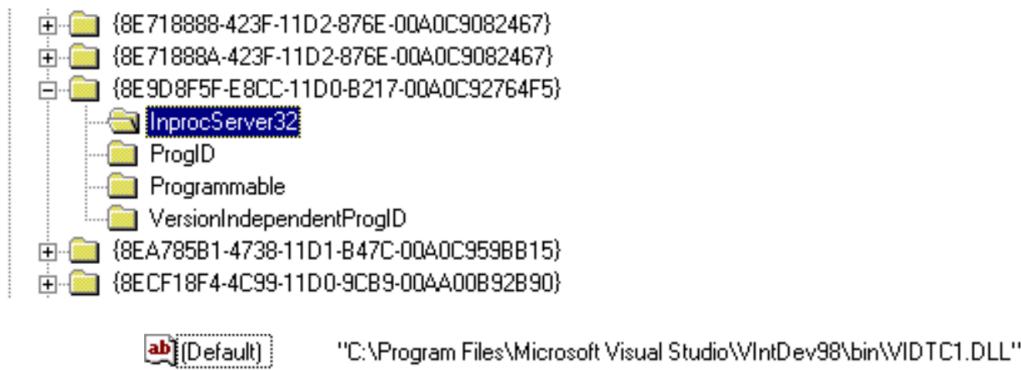
6.3.1 Identificazione degli Oggetti

- Un oggetto (contenitore o meno) può contenere un **riferimento univoco** a un altro oggetto
- Come è possibile **identificare univocamente** un oggetto per poterlo associare a un altro?
- Nel caso di **strutture dati interamente contenute nello spazio di indirizzamento** dell’applicazione, un oggetto può essere identificato univocamente mediante il suo **indirizzo** (logico) **di memoria**
- Nel caso di **database** o di **sistemi distribuiti**, a ogni oggetto deve essere associato un **identificatore univoco persistente** tramite il quale deve essere possibile risalire all’oggetto stesso, sia che risieda in memoria, su disco o in rete
- L’identificatore univoco è un attributo che al momento della creazione dell’oggetto viene inizializzato con:
 - un valore generato automaticamente dal sistema
 - il valore della chiave primaria di una tabella relazionale, ...
- Il nome di tale attributo **potrebbe** essere
 - idDocente

- idStudente, ...

6.3.1.1 Un Esempio Reale

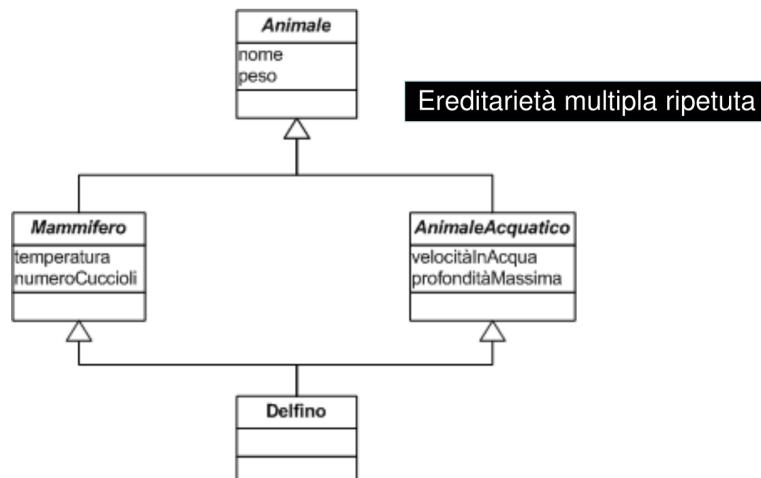
- La **tecnologia COM** (MS) permette a un'applicazione di trovare, caricare e utilizzare *run-time* i **componenti** necessari per la sua esecuzione
- Ogni componente è memorizzato in una DLL (**Dynamic Link Library**) - un file locale o remoto
- Quando l'applicazione ha bisogno di un componente, **il sistema deve essere in grado di localizzare la DLL** che contiene quel particolare componente
- L'indipendenza dalla collocazione fisica non consente di utilizzare un indirizzo fisico (*pathname*)
- Pertanto, deve essere utilizzato un **meccanismo di indirizzamento logico** che permetta di identificare univocamente il file che contiene il componente
- Si utilizzano degli **identificatori globali (GUID = Globally Unique Identifier)**
- Il concetto di GUID è stato introdotto, con un nome leggermente diverso (**UUID = Universally Unique Identifier**), dall'OSF (Open Software Foundation) nelle specifiche **DCE** (Distributed Computing Environment)
- In DCE gli UUID vengono utilizzati per identificare i destinatari delle chiamate di procedura remota (RPC)
- Un GUID è un numero di 128 bit (16 byte) generato in modo da garantire l'unicità nello spazio e nel tempo: MAC (48/64 bit) + ticks (64 bit - 100ns) rappresentato così:
`{32bb8320-b41b-11cf-a6bb-0080c7b2d682}`
- COM utilizza diversi tipi di GUID
- Il tipo più importante di GUID serve a identificare le classi di componenti: ogni classe di componenti COM è caratterizzata da un proprio identificatore che viene chiamato **CLSID** (Class Identifier)
- Disponendo di un CLSID, un'applicazione può chiedere alla funzione di sistema **CoCreateInstance** di creare un'istanza del componente e di restituire un riferimento nel spazio di indirizzamento dell'applicazione stessa
- Il database di sistema di Windows (**registry**) mantiene una corrispondenza tra CLSID ed entità fisiche (DLL, EXE) che contengono l'implementazione dei componenti (server)
- **CoCreateInstance** provvede a
 - reperire il server tramite il *registry*
 - caricarlo in memoria (se non è già presente)
 - creare un'istanza e restituirne un riferimento



- In .NET esiste la classe `System.Guid` che permette di gestire istanze di GUID
- Ad esempio, per ottenere un nuovo GUID, è sufficiente invocare il metodo statico `Guid.NewGuid()` che, ovviamente, restituisce un `System.Guid`
- Altri metodi e operatori permettono di confrontare GUID

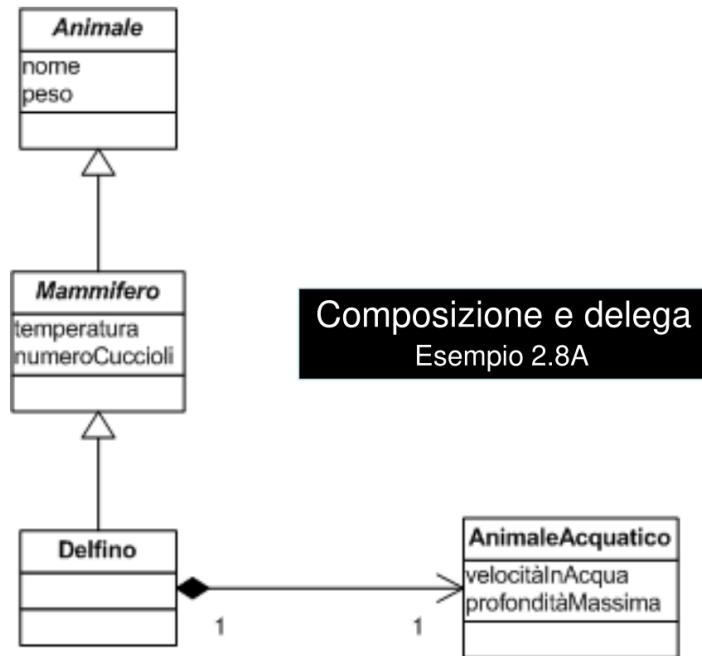
6.3.1.2 Modifiche per Utilizzare il Livello di Ereditarietà Supportato

- Se esistono strutture con ereditarietà multipla
- Se il linguaggio di programmazione non ammette l'ereditarietà multipla
- È necessario convertire le strutture con ereditarietà multipla in strutture con solo ereditarietà semplice



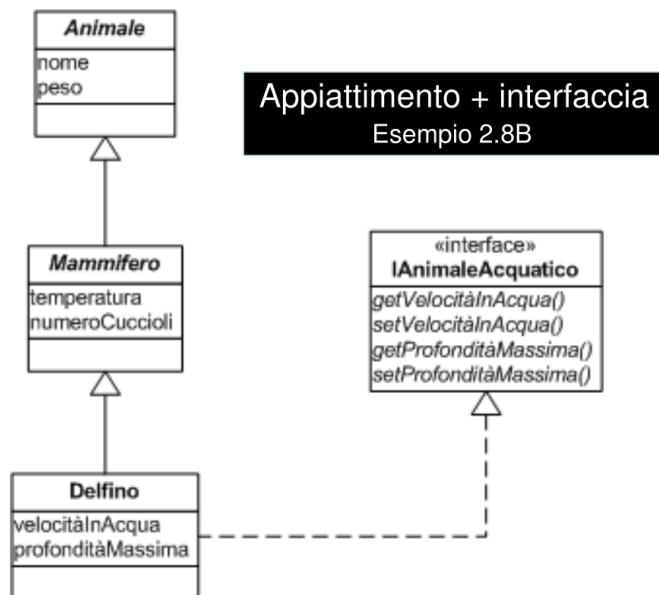
6.3.1.2.1 1^a possibilità (composizione e delega)

- Scegliere la più significativa tra le superclassi ed ereditare esclusivamente da questa
- Tutte le altre superclassi diventano possibili “ruoli” e vengono connesse mediante composizione
- Le caratteristiche delle superclassi escluse vengono incorporate nella classe specializzata tramite composizione e delega e non tramite ereditarietà

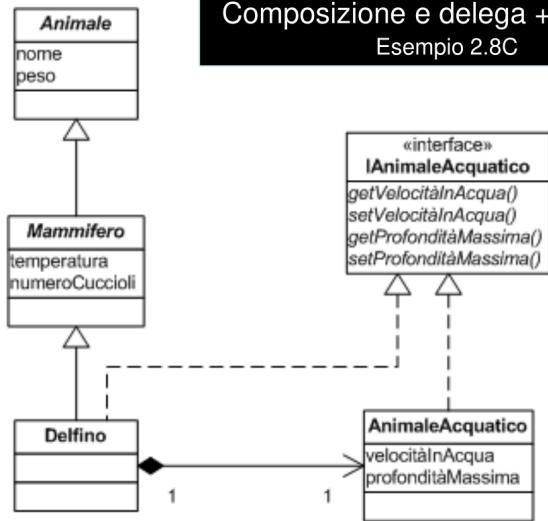


6.3.1.2.2 2^a possibilità (interfaccia)

- Appiattire tutto in una gerarchia semplice e implementare un'interfaccia
- In questo modo, una o più relazioni di ereditarietà si perdono e **gli attributi e le operazioni corrispondenti devono essere ripetuti nelle classi specializzate**



Composizione e delega + interfaccia
Esempio 2.8C



6.4 Miglioramento delle Prestazioni

- Il software con le prestazioni migliori
 - fa la cosa giusta “abbastanza velocemente” (cioè, soddisfacendo i requisiti e/o le attese del cliente)
 - pur rimanendo entro costi e tempi preventivati
- Per migliorare la velocità percepita può bastare
 - la **memorizzazione di risultati intermedi**
 - un’**accurata progettazione dell’interazione con l’utente** (ad es. utilizzando multi-threading)
- Un **traffico di messaggi molto elevato tra oggetti** può invece richiedere dei cambiamenti per aumentare la velocità
- Di norma, la soluzione è che un oggetto possa accedere direttamente ai valori di un altro oggetto (aggirando l’incapsulamento!)
 - Utilizzare metodi inline
 - Utilizzare la dichiarazione friend
 - Combinare insieme due o più classi
- Questo tipo di modifica deve essere presa in considerazione solo dopo che tutti gli altri aspetti del progetto sono stati soggetti a misure e modifiche
- L’unico modo per sapere se una modifica contribuirà in modo significativo a rendere il software “abbastanza veloce” è tramite le misure e l’osservazione