Manutenzione e Reverse Engineering

Riferimenti

- Sommerville, Ingegneria del Software, 8a ed., Capitolo 21

Ulteriori Letture Raccomandate

- Grady Booch, Nine Things you can do with old software, IEEE Software, Sept/Oct 2008
- Canfora, Di Penta “Frontiers of Reverse Engineering: a Conceptual Model”, FOSM08 – IEEE Comp. Soc. 2008

La Manutenzione del software- generalità

- Qualsiasi software, dopo il rilascio della prima release, avrà bisogno di essere modificato.
- I sistemi software sono, per loro natura, sistemi evolutivi che cambiano durante la loro vita.
- Ciò deriva dal fatto che, durante la vita del sistema, le caratteristiche che lo definiscono cambiano, cambiando sia le esigenze di chi usa il software, sia dell'ambiente del mondo reale in cui il sistema opera.
  - Più I requisiti del sistema sono instabili, o specificano un problema in maniera incompleta ed approssimativa, più il sistema avrà bisogno di cambiare.

- L'evoluzione di un software è dunque inevitabile!

Motivazioni per il cambiamento

- Errori possono essere individuati e devono essere corretti;
- Il dominio del software può evolvere;
- Nuovi requisiti possono emergere dopo il rilascio;
- Nuove tecnologie hardware e software possono affermarsi nel frattempo;
- Può essere necessario migliorare la qualità del software (ad esempio l'affidabilità o le performance).

L’importanza dell’evoluzione

- Le organizzazioni proprietarie hanno fatto grossi investimenti per I loro sistemi software, che sono risorse critiche!
- Per preservare il valore di tali risorse, I sistemi devono necessariamente cambiare ed evolvere.
- La manutenzione è un’attività costosa e la maggior parte del budget speso per il software è in genere speso per la sua manutenzione, piuttosto che per il suo sviluppo.

I costi della manutenzione

- Un tipico progetto di sviluppo software mediamente dura tra 1 e 2 anni, mentre…

- La durata del periodo di manutenzione può variare tra 5 e 6 anni [1]
  - Più della metà dei costi di un progetto software sono spesi per la manutenzione
  - Recenti survey riportano la regola dell' 80-20, ossia 80% di sforzo speso per la manutenzione e 20% per lo sviluppo.

- [1] Parikh and Zvegintzov, Tutorial on Software Maintenance, IEEE, 1993

Evoluzione o Declino?

- Fino a che punto si può continuare a far evolvere un sistema software?
- Quando si deve decidere di gettare il vecchio sistema e sostituirlo con uno nuovo?

- Alcune domande da porsi:
  - Il costo di manutenzione è troppo alto?
  - L'affidabilità del sistema è inaccettabile?
  - Non si riesce più ad adattare il software in tempi accettabili?
  - Le prestazioni sono inaccettabili?
  - Le funzionalità del sistema sono poco utili?
  - Ci sono altri sistemi che fanno lo stesso lavoro meglio, più velocemente ed economicamente?
  - Il costo di manutenzione dell'hardware è diventato tale da giustificare la sostituzione con nuovo hardware?

Leggi dell'evoluzione del software

- Proposte da Lehman e Belady a partire dal 1976, in seguito a studi empirici basati inizialmente sull'osservazione dell'evoluzione di 4 versioni successive di un S.O. IBM
  - Modifiche continue
  - Complessità crescente
  - Evoluzione dei grandi sistemi
  - Stabilità organizzativa
  - Conservazione della familiarità

  leggi iniziali
  - Crescita continua
  - Qualità deteriorata
  - Sistema feedback

  leggi più recenti

Le Leggi di Lehman (1974)

- Modifiche continue
  - Un sistema deve necessariamente cambiare, o diventerà progressivamente inutile.
- Complessità crescente
  - Quando un sistema viene modificato, la sua struttura si deteriora: per evitare ciò bisogna investire sulla manutenzione preventiva.
- Evoluzione dei grandi sistemi
  - I grandi sistemi hanno una loro dinamica che è caratterizzata da comportamenti e trends regolari che possono essere usati per fare predizioni: in particolare, la dimensione, il tempo fra due release successive, il numero di errori rilevati sono approssimativamente invarianti per ogni release.

Applicabilità delle Leggi di Lehman

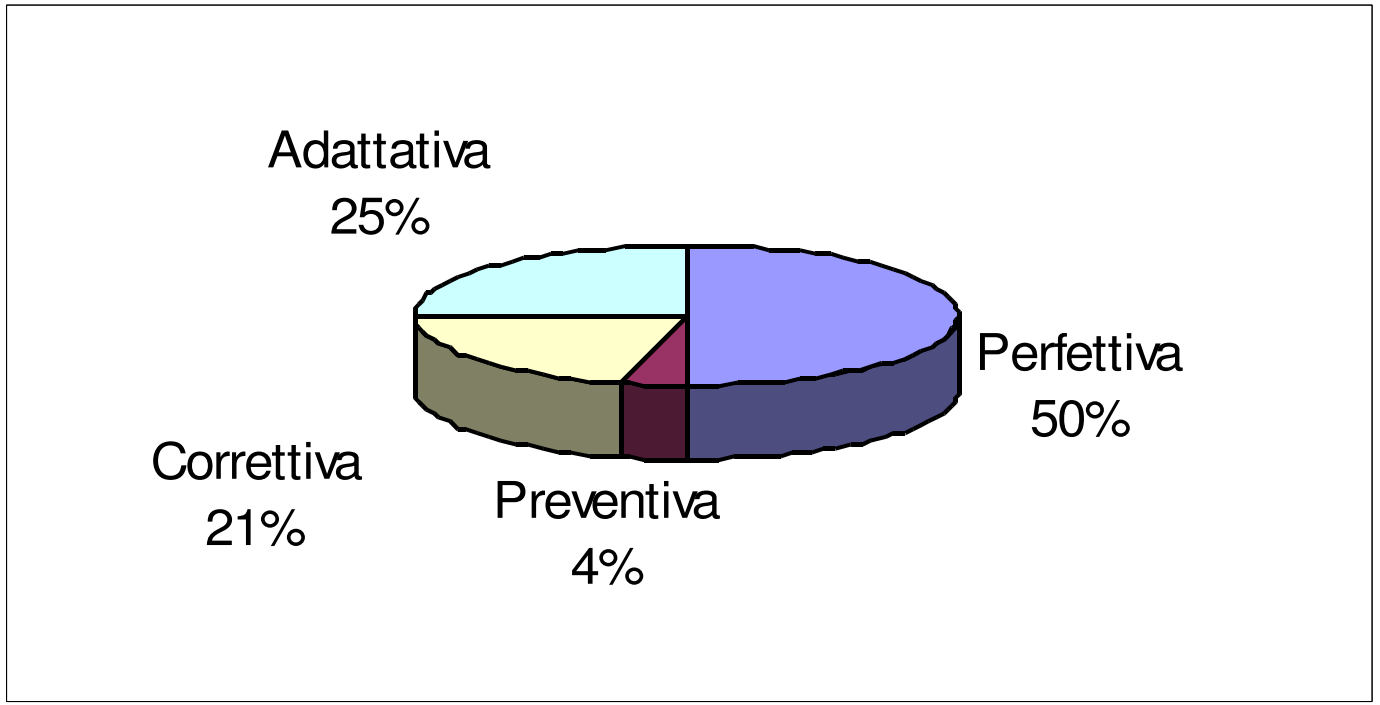
- Le leggi di Lehman sembrano applicabili a sistemi di grandi dimensioni e personalizzati, sviluppati da vaste organizzazioni.

- Non è chiaro come dovrebbero essere modificate per:
  - Sistemi ottenuti attraverso wrapping;
  - Sistemi che incorporano molti componenti COTS;
  - Piccole organizzazioni;
  - Sistemi di medie dimensioni.

Tipi di Manutenzione del software

- Classificazione degli interventi di manutenzione
  - Manutenzione correttiva
    - Modifiche per correggere difetti
  - Manutenzione adattativa
    - Modifiche per adattare il software a cambiamenti dell'ambiente operativo (hardware, software di base, interfacce, organizzazione, legislazione, ecc.)
  - Manutenzione perfettiva
    - Estensione dei requisiti funzionali, o migliorie di requisiti non funzionali in risposta a richieste dell'utente
  - Manutenzione preventiva
    - Modifiche che rendono più semplici le correzioni, gli adattamenti e le migliorie
- Manutenzione di emergenza
    - Manutenzione correttiva non programmata, necessaria a mantenere il sistema in funzione

Distribuzione dello sforzo di manutenzione [1]



[1] Lientz, Swanson, Problems in application software maintenance, 1981 Communication of the ACM

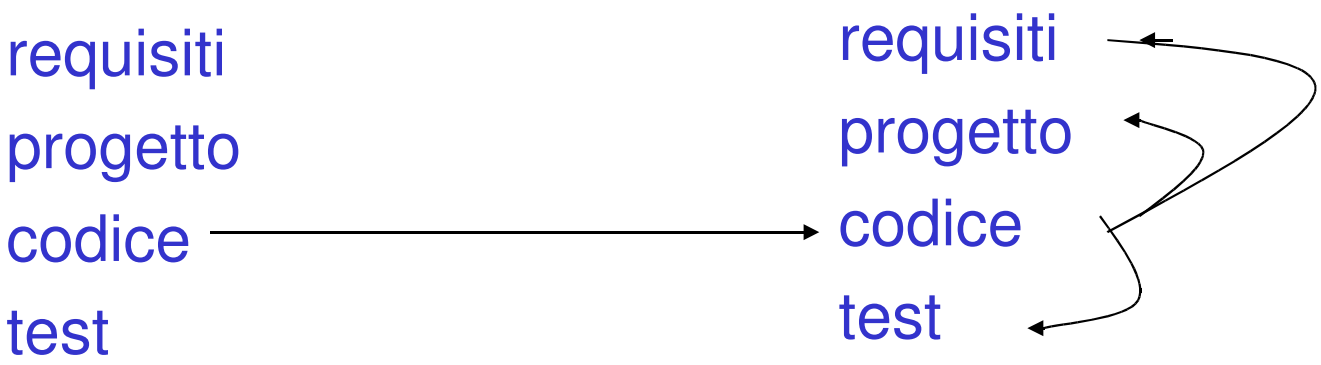
Manutenzione “d’urgenza”

- In alcuni casi le richieste di manutenzione devono essere soddisfatte rapidamente:
  - Se un difetto serio deve essere riparato;
  - Se modifiche dell’ambiente operativo causano effetti collaterali imprevisti sull’operatività del sistema;
  - Se è necessario l’adeguamento urgente a seguito di cambiamenti imprevisti (es. Ambiente o mercato)

- In questo caso, le fasi di analisi e progetto della modifica potrebbero non essere eseguite, implementando direttamente il cambiamento.

Processo di riparazione d'urgenza (quick- fix model)

Vecchio sistema **Nuovo sistema**



- Con tale approccio, la qualità complessiva del software si ridurrà.
- Utile pianificare interventi di aggiornamento della documentazione e/o di miglioramento della qualità del software.

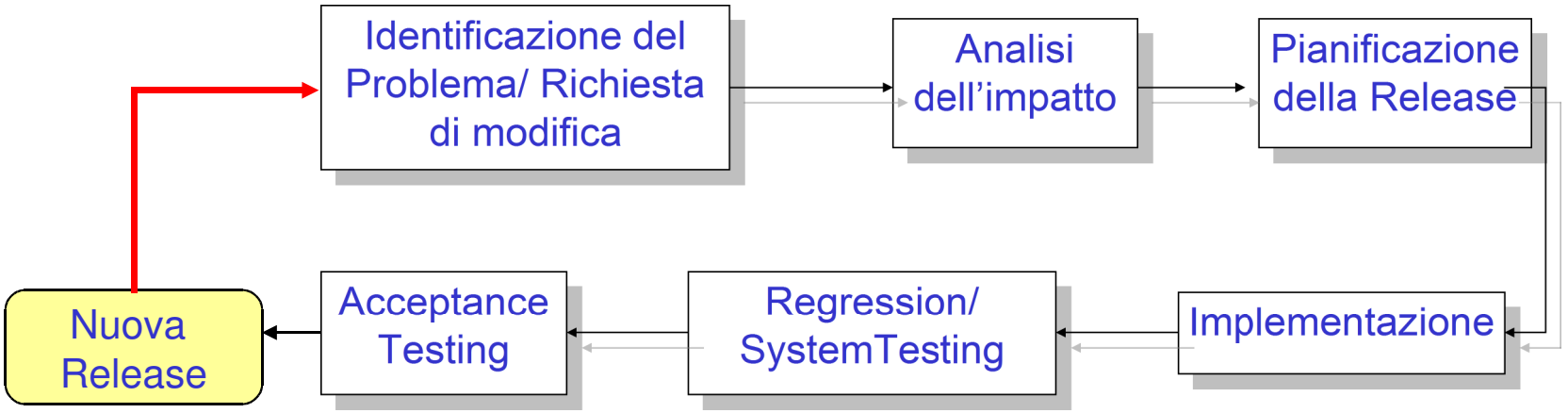
Problemi della manutenzione

- In gran parte dipendono dalla mancanza di controllo e disciplina nelle fasi di analisi e progetto del Ciclo di Vita del Software
- Alcuni fattori tecnici:
  - difficoltà nel comprendere un programma scritto da altri
  - mancanza di documentazione completa/ consistente
  - software non progettato per modifiche future
  - difficoltà nel tradurre una richiesta di modifica di funzionamento del sistema in una modifica del software
  - valutazione dell'impatto di ciascuna modifica sull'intero sistema
  - la necessità di ritestare il sistema dopo le modifiche
  - la gestione della configurazione del software

Fattori di Costo della manutenzione

- Stabilità del team
  - I costi di manutenzione si riducono se lo stesso staff si occupa della manutenzione per lungo tempo
- Responsabilità contrattuale
  - Sviluppo e manutenzione sono talvolta appaltati ad aziende diverse, cosicchè chi sviluppa non ha interesse a semplificare il lavoro di chi effettuerà la manutenzione
- Capacità dello staff
  - Chi si occupa della manutenzione potrebbe non avere lo stesso livello di esperienza e pratica di chi lo ha sviluppato
- Età e struttura del programma
  - Gli interventi di manutenzione tendono a far deteriorare la qualità del software e quindi a rendere più difficili tutti gli interventi di manutenzione necessari
    - v. anche G. Visaggio, “Aging of a legacy system: symptoms and remedies”, Journal of Software Maintenance, Wiley

Il processo di evoluzione del software



Gli interventi per ‘Ringiovanire’ il software

- Sono finalizzati a migliorare la manutenibilità di un software ormai deteriorato dagli interventi di manutenzione subiti.
- Diverse tipi di intervento possibili:

- Ridocumentazione
- Restructuring (o Refactoring)
- Reengineering
- Reverse Engineering

Ridocumentazione

- Consiste in una **analisi statica** del codice sorgente (attraverso appositi strumenti) al fine di produrre documentazione del sistema. Si analizzano:
  - usi delle variabili, chiamate fra componenti, path del flusso di controllo, dimensioni dei componenti, parametri di chiamate, .. Per capire cosa fa il codice e come lo fa.

- Gli output di una attività di ridocumentazione possono essere:
  - grafi delle chiamate,tabelle delle interfacce delle funzioni, dizionari dati, diagrammi del data-flow o control-flow, pseudo-codice, cross-reference fra componenti o variabili
  - Tali output si possono usare per verificare se il software ha bisogno di ristrutturazione.

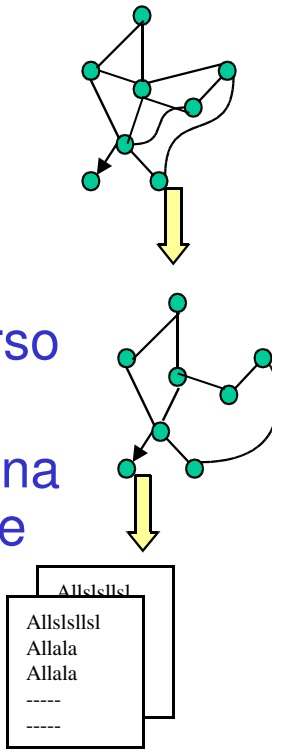
Ridocumentazione

- Un approccio diverso alla ridocumentazione può passare per l'analisi dinamica
  - Si eseguono scenari dei casi d'uso dell'applicazione

- Per analisi dinamica si possono ottenere:
  - Modelli dell'interfaccia utente
  - Modelli dell'interazione tra i componenti
  - Documentazione per l'utente finale
    - Tutorial, …

Restructuring

- Attività che trasforma il codice esistente in codice equivalente dal punto di vista funzionale, ma migliorato dal punto di vista della sua qualità.
- In genere si esegue in tre passi:
  1. Analisi statica del codice per ottenerne una rappresentazione interna (es. call graph, control-flow graph…)
  2. Semplificazione della rappresentazione interna attraverso tecniche di trasformazione automatiche.
  3. La nuova rappresentazione viene usata per generare una versione strutturata (migliorata) ed equivalente al codice originario.



Refactoring

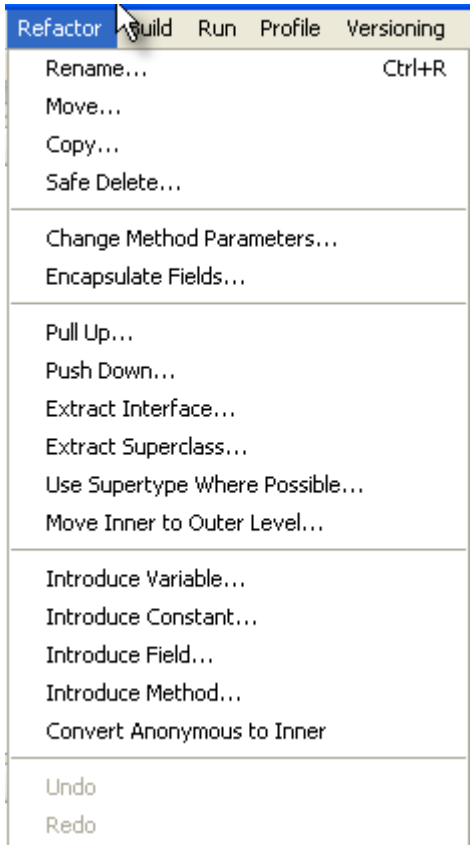
- Anche i sistemi object-oriented sono soggetti al deteriorarmento e diventano legacy!
- Il refactoring è un insieme di tecniche usabili per migliorare il codice ed il design di sistemi object-oriented.
  - Martin Fowler è autore del libro “Refactoring: Improving the Design of Existing Code” (1999)dove presenta più di 70 pattern per eseguire il refactoring.
- Tali tecniche cercano di eliminare i cosiddetti *Bad Smells* dal codice.

Esempi di Bad Smells nel codice

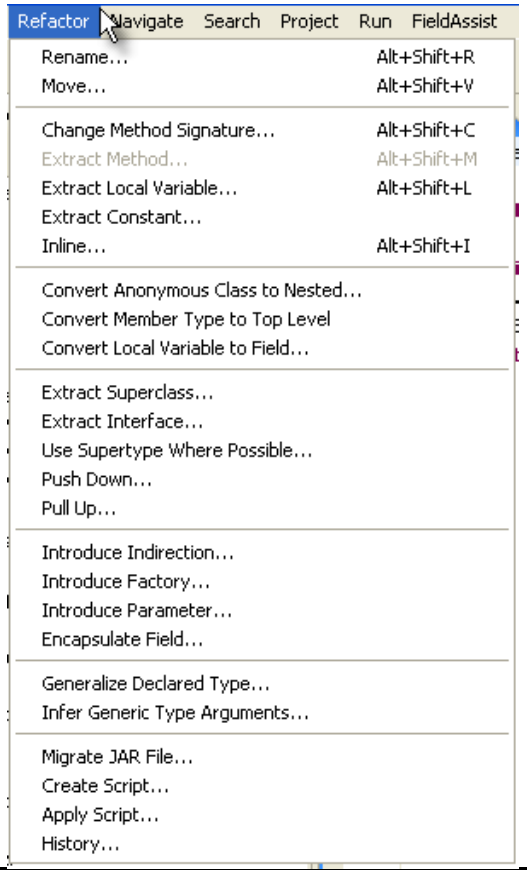
Problema (Bad Smell)	**Pattern di Refactoring applicabile**
Codice duplicato	*Extract Method*
Metodi troppo lunghi	*Extract Method*
Classi troppo grandi	*Extract Class, Extract Sub-Class, Extract Interface*
Lunghe liste di parametri di metodi	*Replace parameter with Method*
Cambiamenti divergenti in una classe (una classe è soggetta a cambiamenti per tanti motivi)	*Extract Class*
…	…

Esempi di Tool per il Refactoring

Net Beans Refactoring



Eclipse Refactoring



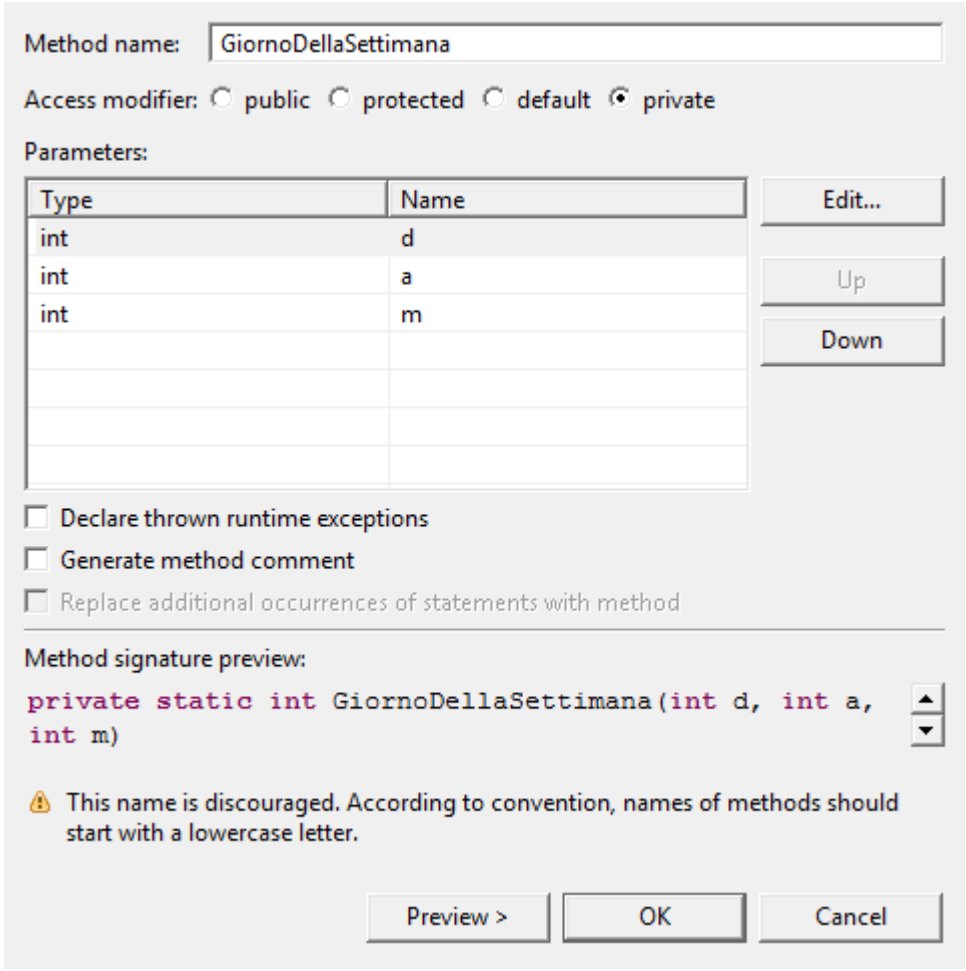
Esempi di Refactoring: extract method

- Extract method
  - Seleziona un pezzo di codice
  - Imposta nome e parametri
  - Sostituisci

```
if ((m==4 || m==6 || m==9 || m==11) && d>30)
    return "Errore";

if (m<=2)
 {
  m = m + 12;
  a--;
 };
 int f1 = a / 4;
 int f2 = a / 100;
 int f3 = a / 400;
 int f4 = (int) (2 * m + (.6 * (m + 1)));
 int f5 = a + d + 1;
 int x = f1 - f2 + f3 + f4 + f5;
 int k = x / 7;
 int n = x - k * 7;

 if (n==1)
     return "Lunedi";
```

Esempi di Refactoring: extract method

- Extract method
  - Seleziona un pezzo di codice
  - Imposta nome e parametri
  - Sostituisci



Esempi di Refactoring: extract method

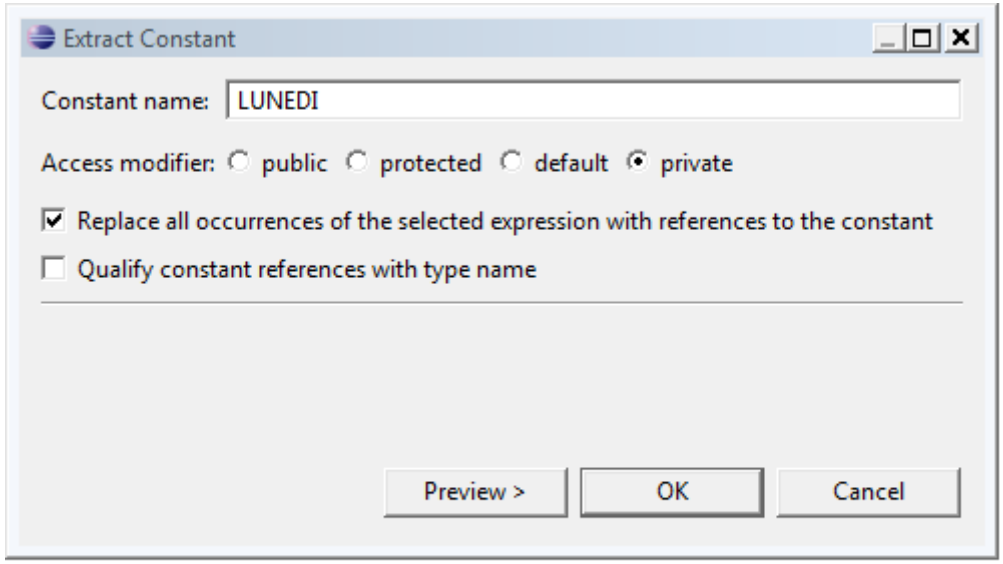
- Extract method
  - Seleziona un pezzo di codice
  - Imposta nome e parametri
  - Sostituisci

```
int n = giornoDellaSettimana(d, a, m);
```

```
private static int giornoDellaSettimana(int d,
int a, int m) {
if (m<=2)
 {
  m = m + 12;
  a--;
 };
 int f1 = a / 4;
 int f2 = a / 100;
 int f3 = a / 400;
 int f4 = (int) (2 * m + (.6 * (m + 1)));
 int f5 = a + d + 1;
 int x = f1 - f2 + f3 + f4 + f5;
 int k = x / 7;
 int n = x - k * 7;
return n;
}
```

Esempi di Refactoring: extract constant

- Extract constant:
  - Seleziona un valore costante
  - Converti in costante (tutte le sue occorrenze)



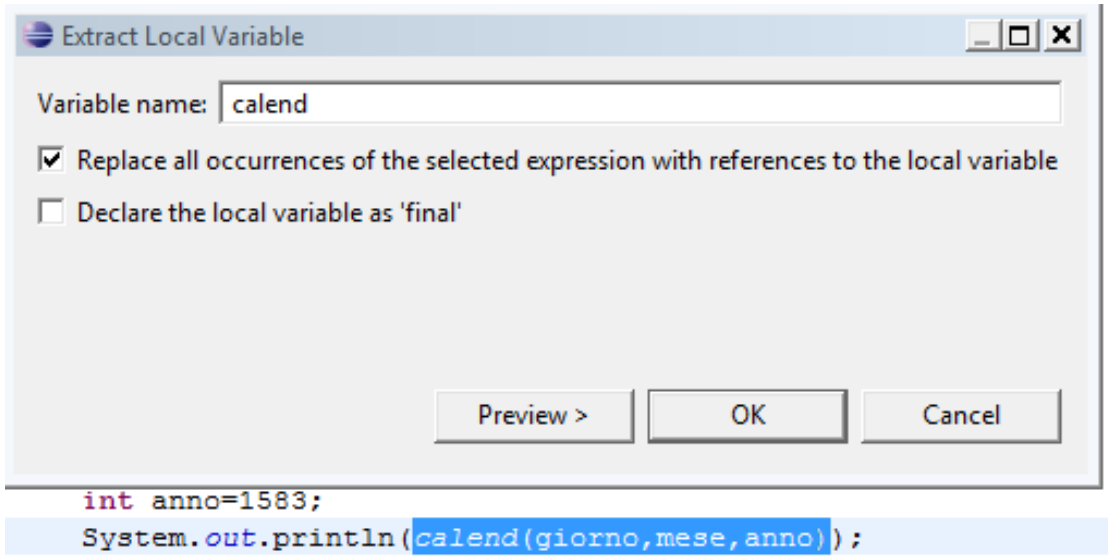
Esempi di Refactoring: extract constant

- Extract constant:
  - Seleziona un valore costante
  - Converti in costante (tutte le sue occorrenze)

```
private static final String
LUNEDI = "Lunedi";

…

 if (n==1)

 return LUNEDI;
```

Esempi di Refactoring: extract local variable

- Extract local variable:
  - Seleziona un'espressione che ha un valore
  - Converti l'espressione in una variabile (tutte le sue occorrenze)



Esempi di Refactoring: extract local variable

- Extract local variable:
  - Seleziona un'espressione che ha un valore
  - Converti l'espressione in una variabile (tutte le sue occorrenze)

```
String calend =
calend(giorno,mese,anno);

System.out.println(calend);
```

Strumenti di analisi del codice e refactoring

- Lint. Analizzatore del codice nativo per Android, introdotto in ADT 16 (gratuito).
- FindBugs. Analizzatore del codice Java.
- Checkstyle. Analizzatore del codice Java.
- CodePro Analytix. Analizzatore del codice Java.
- PMD. Analizzatore del codice Java.
- MOTODEV App Validator. Analizzatore del codice nativo per Android, (gratuito)

Reengineering

- It is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [*Chikofsky and Cross*]

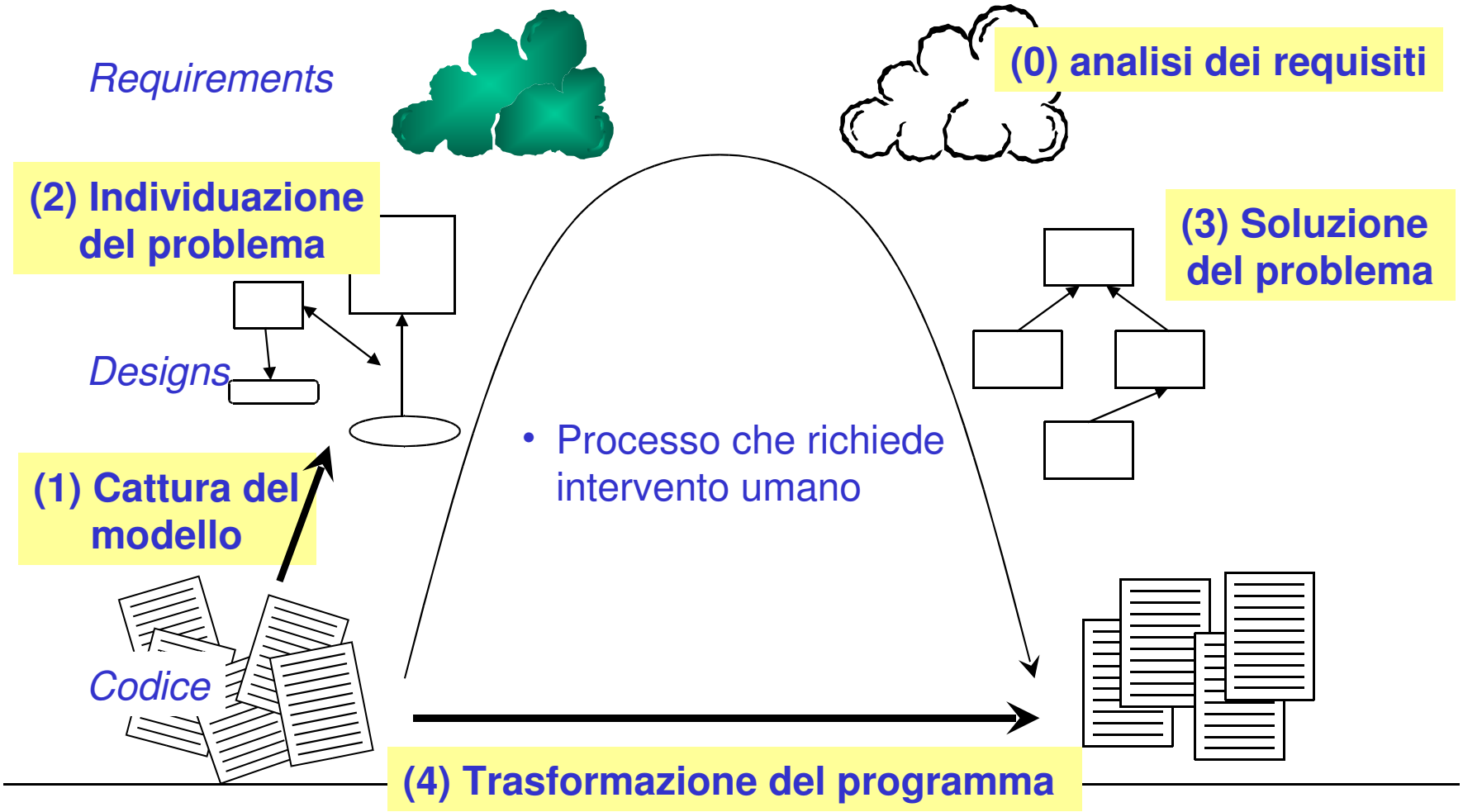
- La reingegnerizzazione (reengineering) è un'attività di re-implementazione di un sistema software svolta per migliorare la manutenibilità di un software esistente.

- Essa può comprendere:
  - Ridocumentazione, Ristrutturazione, Refactoring e riscrittura di parte del software (o anche di tutto) senza modificare l'insieme di funzionalità che esso realizza

Obiettivi del Reengineering

- *Modularizzazione del sistema*
  - Suddivisione di un sistema monolitico in parti da riusare separatamente
- *Miglioramento delle Performance*
  - Migliorare le prestazioni di un sistema esistente
- *Migrazione (o Porting) verso altre Piattaforme*
  - Necessità di localizzare I componenti dipendenti dalla piattaforma
- *Estrazione del progetto*
  - Per migliorare maintainability, portability, etc.
- Migrazione verso una *nuova Tecnologia*
  - quali nuove caratteristiche di un linguaggio, standards, librerie, etc.

Il ciclo di vita del Reengineering



Reengineering vs. Restructuring

- **Reengineering**
  - attività di ristrutturazione a livello della riorganizzazione dell'architettura modulare di un sistema; si parte da una certa organizzazione dei moduli del sistema e del relativo flusso dati, e se ne producono altre con migliori caratteristiche di qualità, come ad esempio, la riduzione dell'accoppiamento fra i moduli, il controllo dell'uso di variabili globali e la riorganizzazione di data repository e così via
- **Restructuring (o Code Refactoring)**
  - attività che trasforma il codice esistente in codice equivalente dal punto di vista funzionale, ma migliorato dal punto di vista della sua qualità

Reverse Engineering

- È un'attività che consente di ottenere specifiche e informazioni sul design di un sistema a partire dal suo codice, attraverso processi di estrazione ed astrazione di informazioni.

- La definizione di **Chikofsky and Cross**, 1990 [1]:
- *Analyzing a subject system*
  - *to identify the system's components and their inter-relationships and*
  - *to create representations of the system in another form or at a higher level of abstraction*
- *A two-steps process*
  - *information extraction*
  - *view abstraction*

[1] **Chikofsky and Cross,** Reverse engineering and design recovery: A taxonomy. *IEEE Software*,1990

Fasi di Estrazione ed Astrazione

- Estrazione
  - Analisi del codice o di altri artifatti software, allo scopo di ottenere informazioni relative al sistema analizzato.
  - Particolarmente utili sono quelli strumenti in grado di estrarre informazioni da un codice sorgente qualsiasi, nota che sia la grammatica del linguaggio di programmazione (ad esempio JavaCC)
- Astrazione
  - Si esaminano le informazioni estratte e si cercano di astrarre diagrammi, o viste, ad un più alto livello di astrazione (es.: diagrammi di progetto, architetturali, del dominio dei dati)
  - I processi di astrazione non sono completamente automatizzabili poichè necessitano di conoscenza ed esperienza umana

Un Modello concettuale per il Reverse Engineering

- Canfora e Di Penta [1] hanno recentemente proposto un modello concettuale che sistematizza tutti i concetti chiave di un processo di reverse engineering, quali:

- Goal del processo
- Artifatti coinvolti nel RE
- Analizzatori usabili
- Viste producibili

- [1] Canfora, Di Penta “Frontiers of Reverse Engineering: a Conceptual Model”, FOSM08 – IEEE Comp. Soc. 2008

Problemi indecibili nel Reverse Engineering

- non è possibile, a partire dal solo codice, astrarre ***il*** *progetto* dal quale esso è stato prodotto
  - non è invece indecidibile il problema di astrarre un progetto coerente con il codice;
- non è possibile, a partire dal solo programma oggetto, astrarre ***il*** *programma* sorgente dal quale esso è stato prodotto
  - non è invece indecidibile il problema di astrarre un programma sorgente che generi il dato programma oggetto.

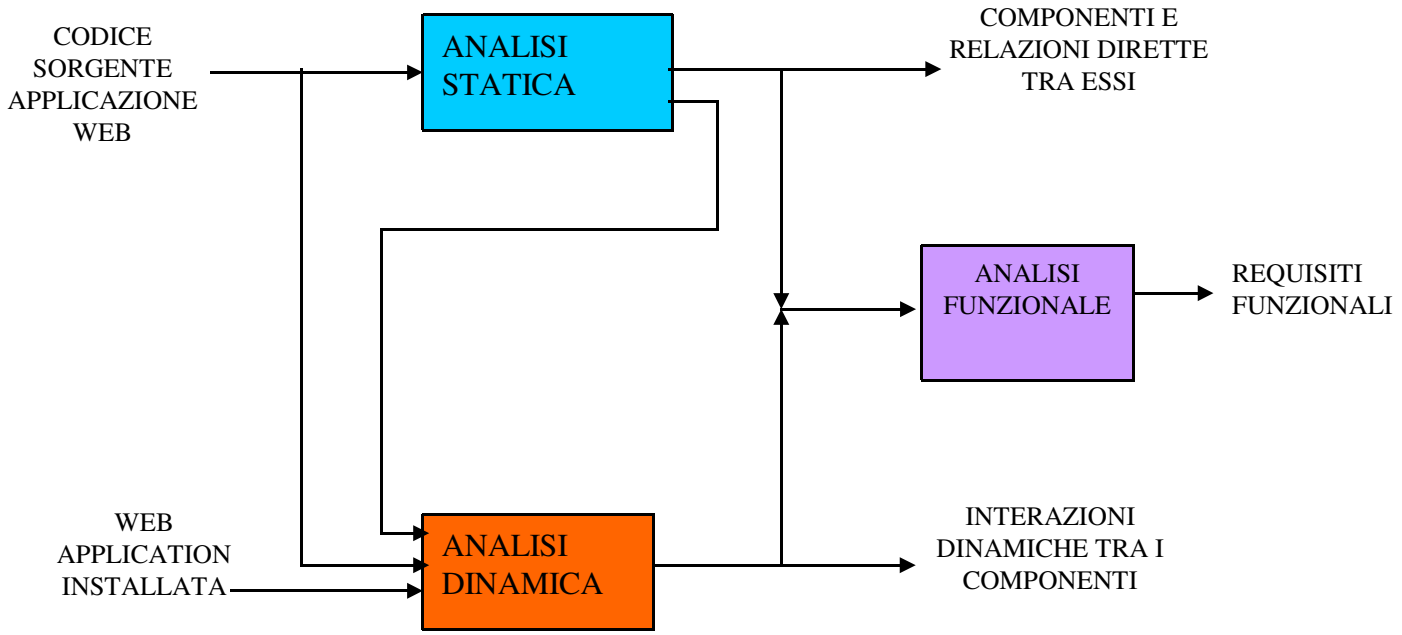
Problemi del Reverse Engineering

- Il processo di produzione del software è costellato di *pozzi* nei quali si perde parte della conoscenza: non tutta la conoscenza ed esperienza messa in campo dall'ingegnere del software in una fase di produzione (ad es. progettazione) viene in qualche forma rappresentata nello stesso prodotto di fase (progetto) o in quello delle fasi successive (ad es. codice).

- Questo comporta che ai problemi di indecidibilità si aggiungono quelli dovuti alla perdita di conoscenza che richiedono, per la realizzazione completa di un'astrazione, l'**aggiunta di conoscenza** ed esperienza da parte dell'ingegnere del software (almeno per ora).

Esempio (d'annata):

Reverse Engineering di applicazioni web

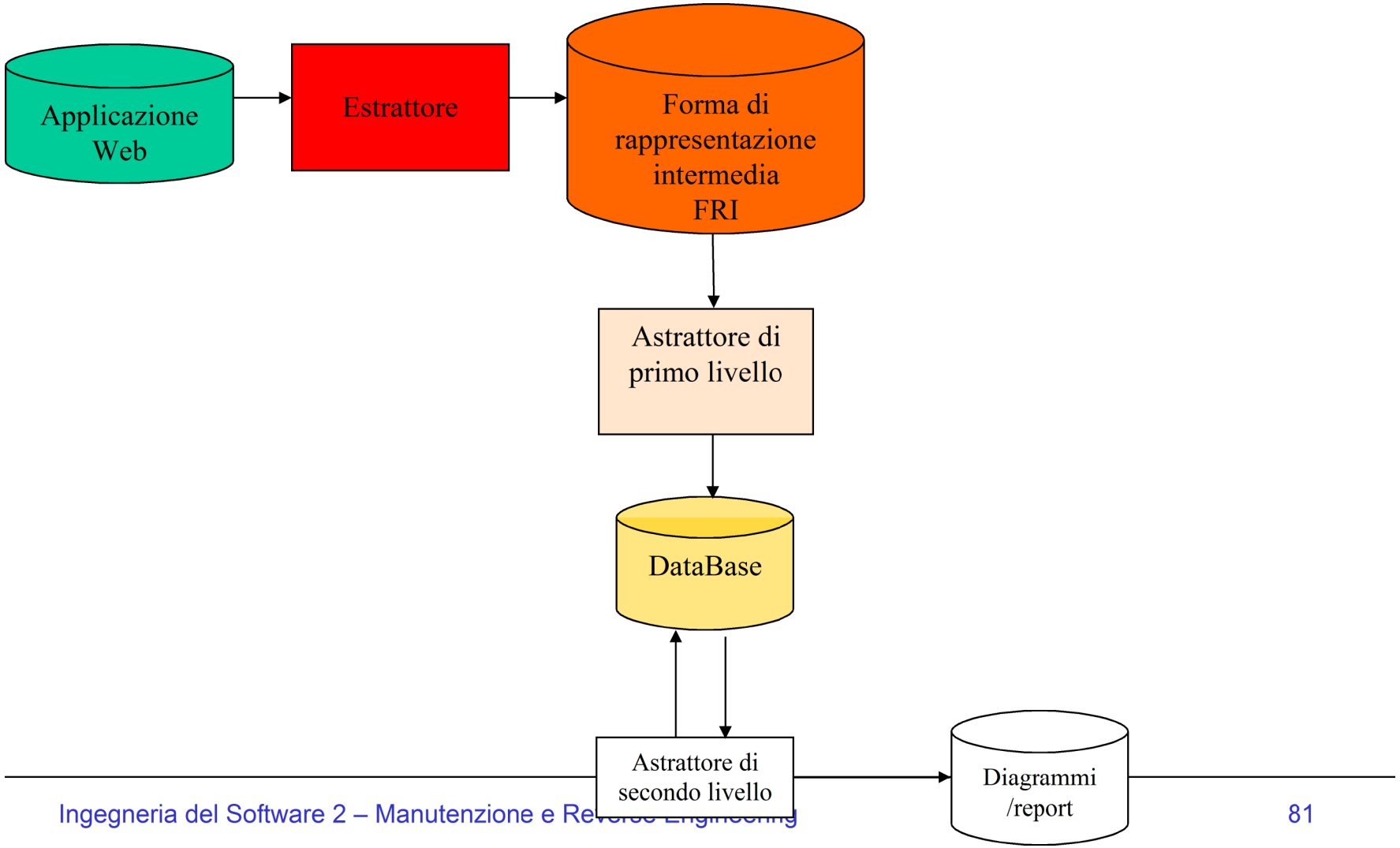


Analisi statica vs Analisi Dinamica

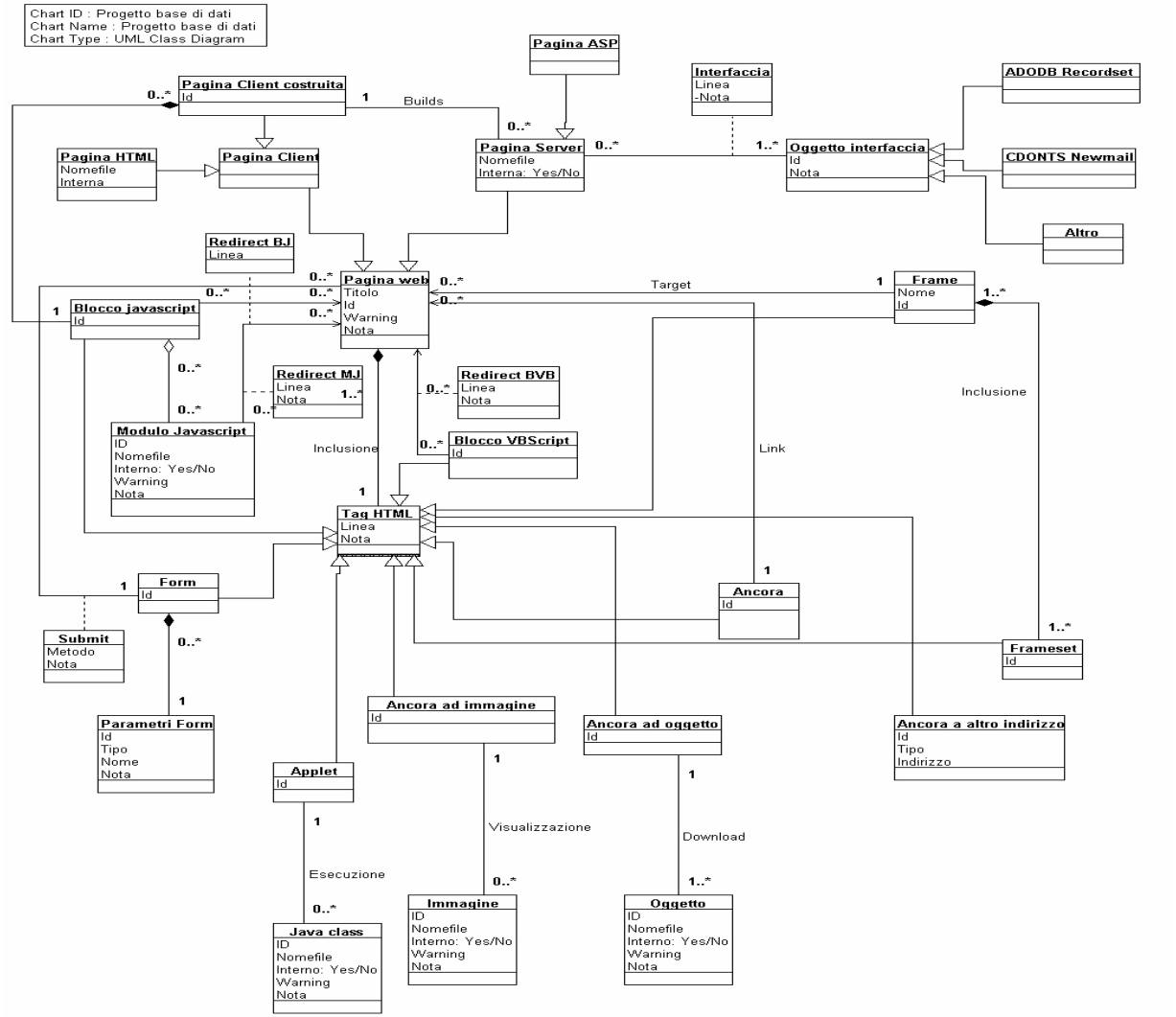
- L'analisi statica:
  - Deve essere effettuata necessariamente sul codice sorgente
    - Possibile solo per li sviluppatori dell'applicazione
  - Estrae solo un sottoinsieme delle informazioni
    - Ad esempio, nei software object oriented non può estrarre gli oggetti istanziati dinamicamente (e nemmeno le eventuali classi dichiarate a tempo di esecuzione)
  - Non va a modificare il codice sorgente

- L'analisi dinamica:
  - Può essere effettuata anche dagli utenti dell'applicazione
  - Potenzialmente può estrarre tutte le interazioni che vengono in essere durante l'esecuzione del software
  - Necessita di sonde da inserire nel codice (per esportare dati)
  - Non ha una terminazione
    - Dovrebbe riferirsi ad un insieme "significativo" di esecuzioni dell'applicazione
      - Ad esempio una test suite che soddisfi un certo criterio di copertura

Un tool di supporto all'analisi statica: Architettura del sistema complessivo



Sviluppo del tool estrattore: Analisi delle informazioni



Sviluppo del tool estrattore:

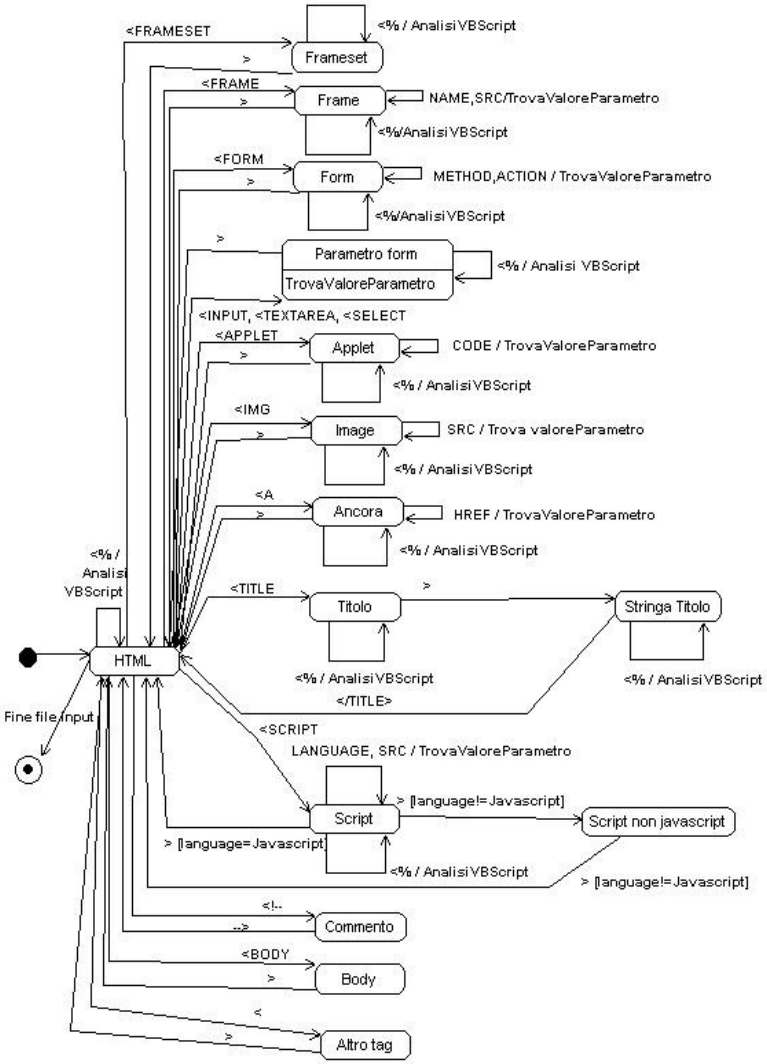
Definizione della forma di rappresentazione intermedia

- Censimento di tutte le espressioni del codice sorgente che devono essere riconosciute: (tag HTML - comandi Javascript -comandi VBScript - eventi di apertura e chiusura file)
- Per ognuna di queste espressioni viene definita la sintassi del tag corrispondente nella forma di rappresentazione intermedia
- Per ogni tag della forma di rappresentazione intermedia viene definita la semantica

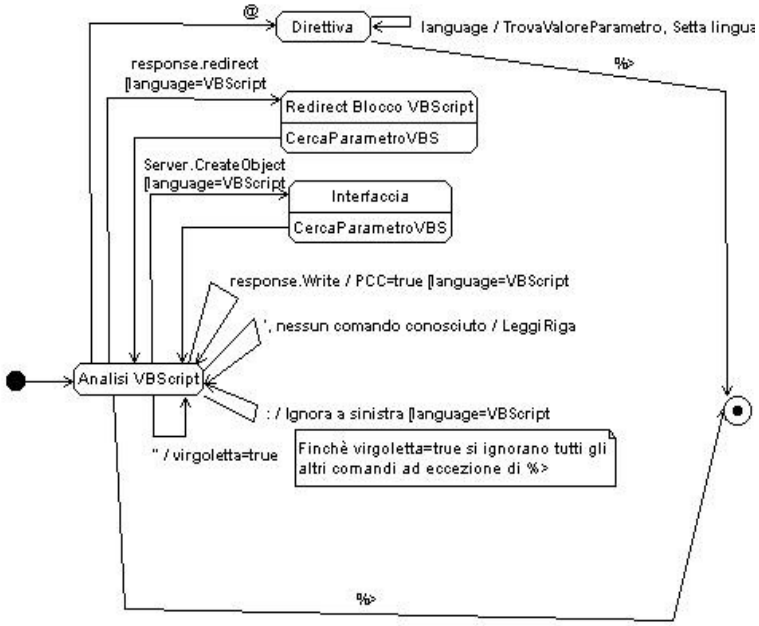
```
<APERTURA>
        <NOMEFILE="\index.htm">
</APERTURA>
<TITOLO>
        <TITOLO="Giuridea - Forum e Laboratorio
Giuridico">
</TITOLO>
<APERTURA BLOCCO JAVASCRIPT>
        <LINEA=22>
</APERTURA BLOCCO JAVASCRIPT>
<CHIUSURA BLOCCO JAVASCRIPT>
<IMMAGINE>
        <LINEA=50>
        <NOMEFILE="images/title.jpg">
</IMMAGINE>
<ANCORA>
        <LINEA=51>
        <NOMEFILE="Archivio/Archivio.htm">
</ANCORA>
```

Sviluppo del tool estrattore: Implementazione

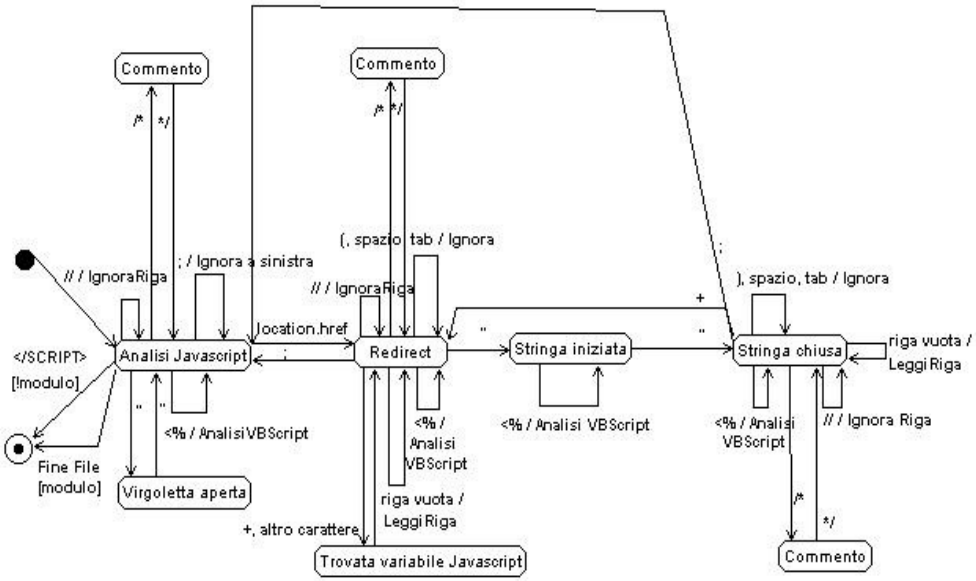
Statechart raffigurante l'automa riconoscitore di tag HTML



Sviluppo del tool estrattore: Implementazione



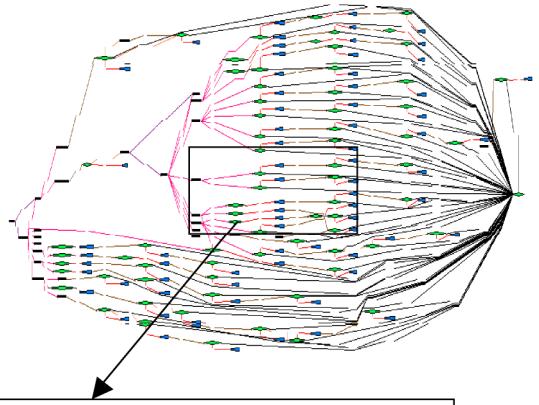
Automa riconoscitore di parole chiave VBScript

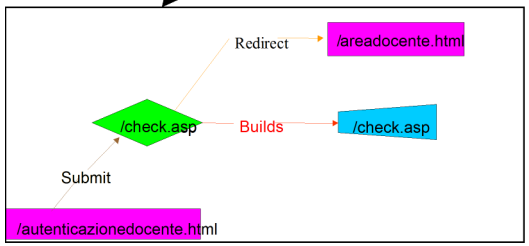


Automa riconoscitore di parole chiave Javascript

Tool astrattore

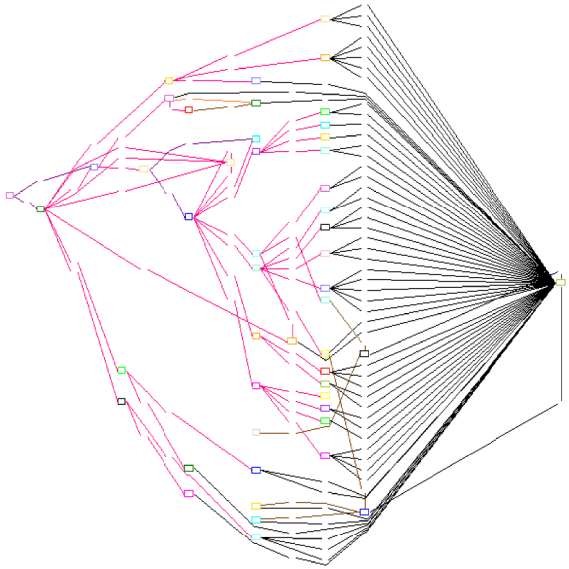
Diagramma delle pagine





Euristiche per il raggruppamento delle pagine

Diagramma dei gruppi di pagine correlate



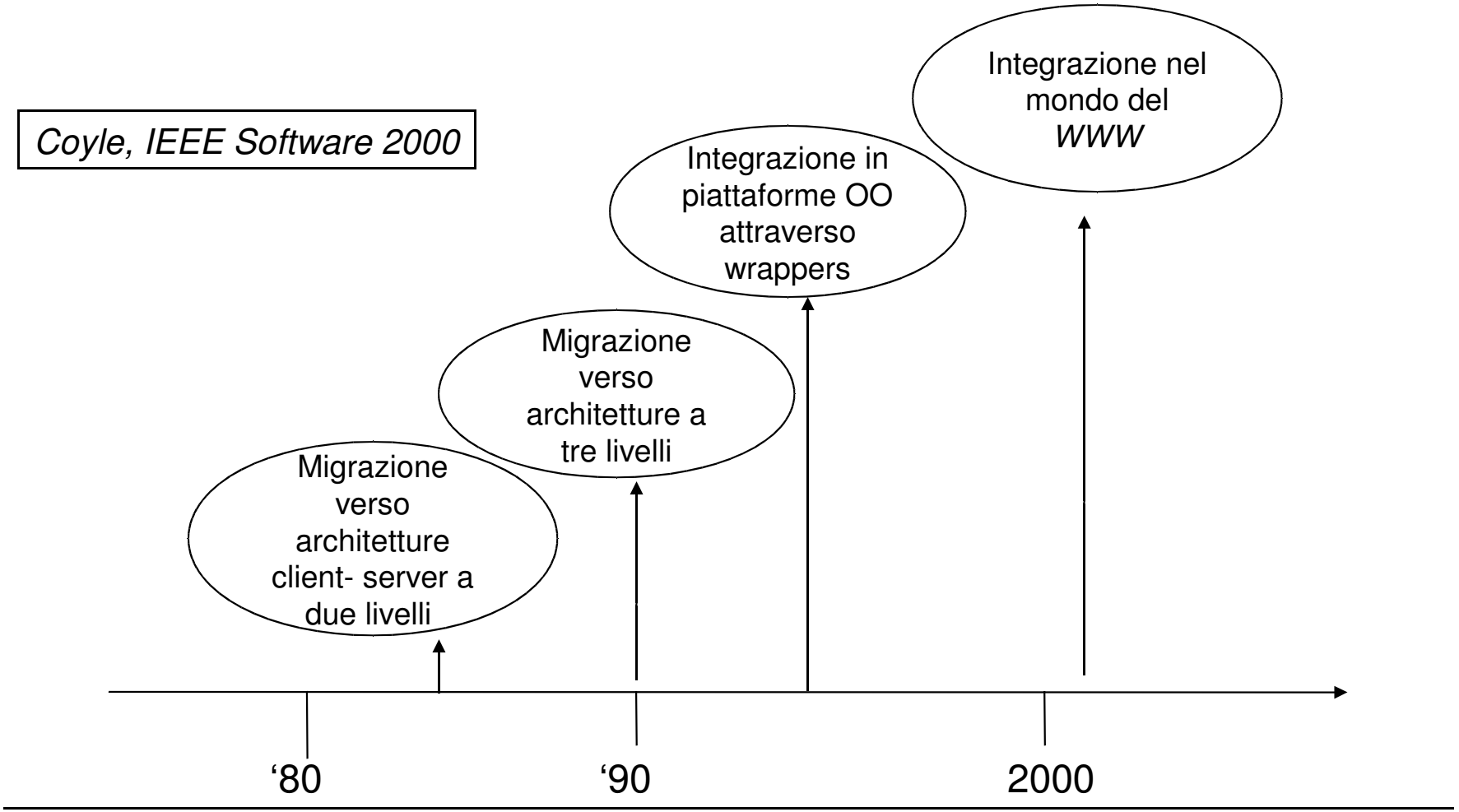
Validazione dei gruppi e loro interpretazione come casi d'uso

Il problema dei Legacy Systems

- Un sistema legacy (“ereditato”)
  - é spesso vecchio (10 anni o più di vita);
  - è di grandi dimensioni (centinaia di migliaia di linee di codice)
  - è scritto in assembler o in un linguaggio di vecchia generazione
  - è stato probabilmente sviluppato prima che si diffondessero i moderni principi dell’ingegneria del software
  - La manutenzione é stata svolta in modo da seguire le modifiche nei requisiti, aumentando così l’entropia (il disordine) del sistema
  - La manutenzione risulta ormai difficile e costosa
  - Realizza funzionalità cruciali e irrinunciabili per l’organizzazione
  - Contiene anni di esperienza accumulata nell’ambito del dominio specifico del problema

La gestione dei Sistemi Legacy: due decenni di strategie

Coyle, IEEE Software 2000



Un sistema legacy obbliga il management a cercare soluzioni e strategie di manutenzione

- Svariate alternative disponibili:
  - Eliminazione del sistema e sviluppo di un nuovo sistema
  - Recupero dei componenti più preziosi del sistema, sostituendo i restanti con prodotti preconfezionati (COTS)
  - Eliminazione dei componenti ormai inutili, del codice obsoleto e dei dati “morti”
  - Congelamento del sistema *as is*, e riutilizzo mediante tecniche di wrapping
  - Manutenzione Ordinaria
  - Manutenzione Preventiva (re-documentation, restructuring o reengineering)
  - Migrazione …

Una recente analisi di strategie proposta da Grady Booch

G. Booch, Nine things you can do with old software, IEEE Software, Sept 2008

- Abbandonarlo
  - quando il suo valore economico si è esaurito, o lo sforzo di risviluppo non è eccessivo
- Regalarlo
  - Se non serve più, si può cederlo a qualche comunità open-source dove potrà ancora tornare utile a qualcuno
- Ignorarlo
  - Se è abbastanza stabile, e fa qualcosa di utile, si può continuare a usarlo senza però modificarlo.

Nove cose da poter fare con software legacy

- Farlo sopravvivere
  - Quando l'hardware su cui gira non è più supportato (e non si dispone del codice sorgente per portarlo in nuove piattaforme), si fa sopravvivere o cercando vecchio hardware da cannibalizzare, o usando emulatori di piattaforma.
- Riscriverlo
  - Quando la manutenzione è troppo costosa, o il sistema è troppo fragile, si può riscriverlo (ma sapendo che ottenere un sistema funzionalmente equivalente al primo è impossibile, e che bisognerà probabilmente convincere gli utenti ad accettare qualche cambiamento)
- Farlo fruttare (in qualche modo)
  - Cercare parti del sistema da conservare (algoritmi, pattern, astrazioni…) perché ancora utili, ed usarle come una base di conoscenza per un nuovo sviluppo

Nove cose da poter fare con software legacy

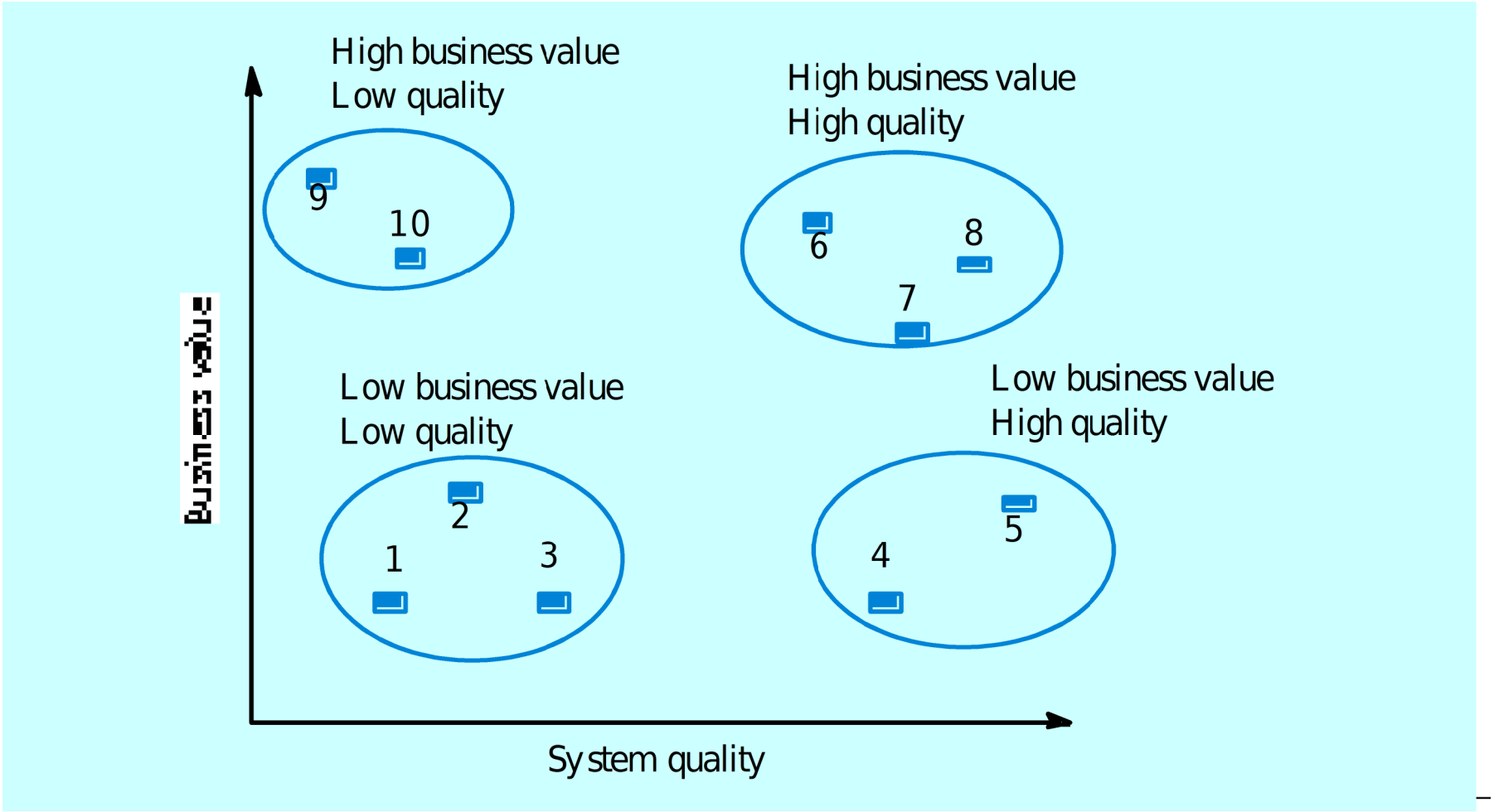
- Wrapping
  - Usare tecniche di wrapping per integrarlo in nuove piattaforme (quali SOA)
- Trasformarlo
  - È la strategia più difficile e va praticata per mantenere il sistema in condizioni ottimali, se si prevede di continuare ad usarlo a lungo termine: va dal semplice refactoring, alla trasformazione architetturale.
- Preservarlo
  - Anche il software antico può avere un valore storico (es. vecchi sistemi operativi, o videogiochi) e culturale da preservare (magari in un Museo della Storia dei Computer)

Conclusione di Booch

- Per Software economicamente interessante ci sono due possibili opzioni di gestione:
- Preservarlo
  - Si fa per motivi soprattutto irrazionali (avversione al rischio)
- Farlo evolvere
  - Quando ciò può contribuire al suo valore a lungo termine

- La scelta è alla fine sempre dettata da fattori economici (e non tecnici)!

System quality and business value



Legacy system categories

1. Low quality, low business value
   - Dovrebbe essere abbandonato
1. Low-quality, high-business value
   - Realizza funzionali importante ma è costoso mantenerlo. Dovrebbe essere reingegnerizzato in modo da rendere le future (necessarie) operazioni di manutenzione più agevoli ed efficaci (cioè finire nel quarto caso)
1. High-quality, low-business value
   - Si può decidere sia di abbandonarlo (in quanto poco importante), sia di rimpiazzarlo con COTS (se realizza qualcosa di generale, indipendente dal dominio specifico) oppure mantenerlo (dato che i costi di manutenzione saranno limitati)
1. High-quality, high business value
   - Su di esso si eseguono le operazioni di manutenzione
     - In questo modo, però, la qualità diventerà via via più bassa, fino a finire nel secondo caso