L’evoluzione del software

Prof. Paolo Ciancarini
Corso di Ingegneria del Software
CdL Informatica
Università di Bologna

Obiettivo della lezione

- Tipi di evoluzione del software
- La manutenzione nel ciclo di vita del sw
- Impatto economico della manutenzione
- Strumenti di manutenzione
- Metriche utili per la manutenzione

Modificare il codice

Per modificare un pezzo di software, bisogna leggerlo

Leggere un programma è più difficile che scriverlo

Modificarlo *senza introdurre errori* è ancora più difficile, se non si hanno i test di regressione

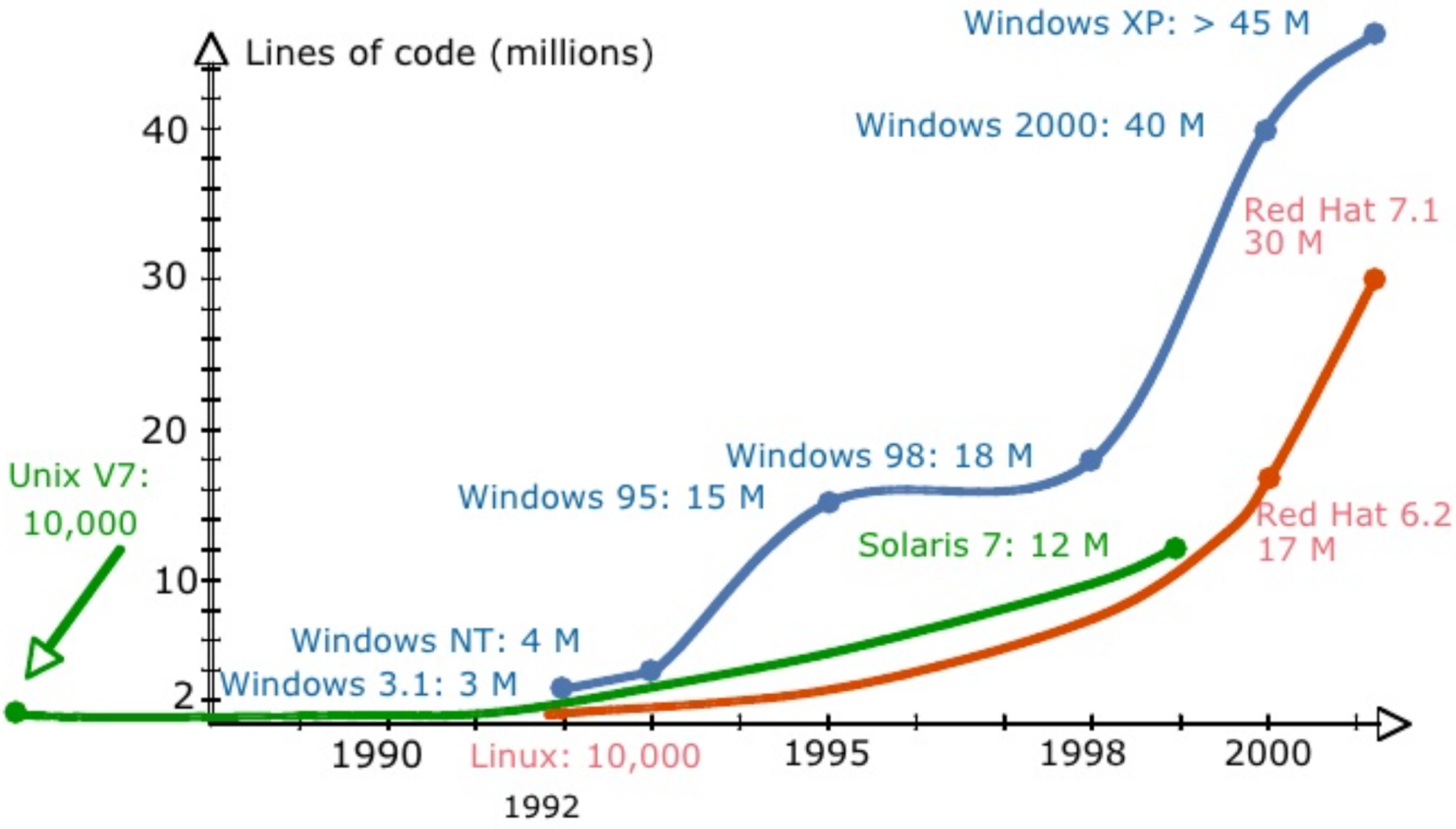
Cosa fa questo codice?

```
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
    else switch(m1-=h){
        case f:
            a=(b=(c=(d=g)<<g)<<g)<<g;
            return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
        case h:

for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
        case g:
            if(n<h)return(g);
            if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
            else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
            if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
            return(o[b-g]%n+k-h);
        default:
            if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
            for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
        }
}
```

Programmazione “offuscata”

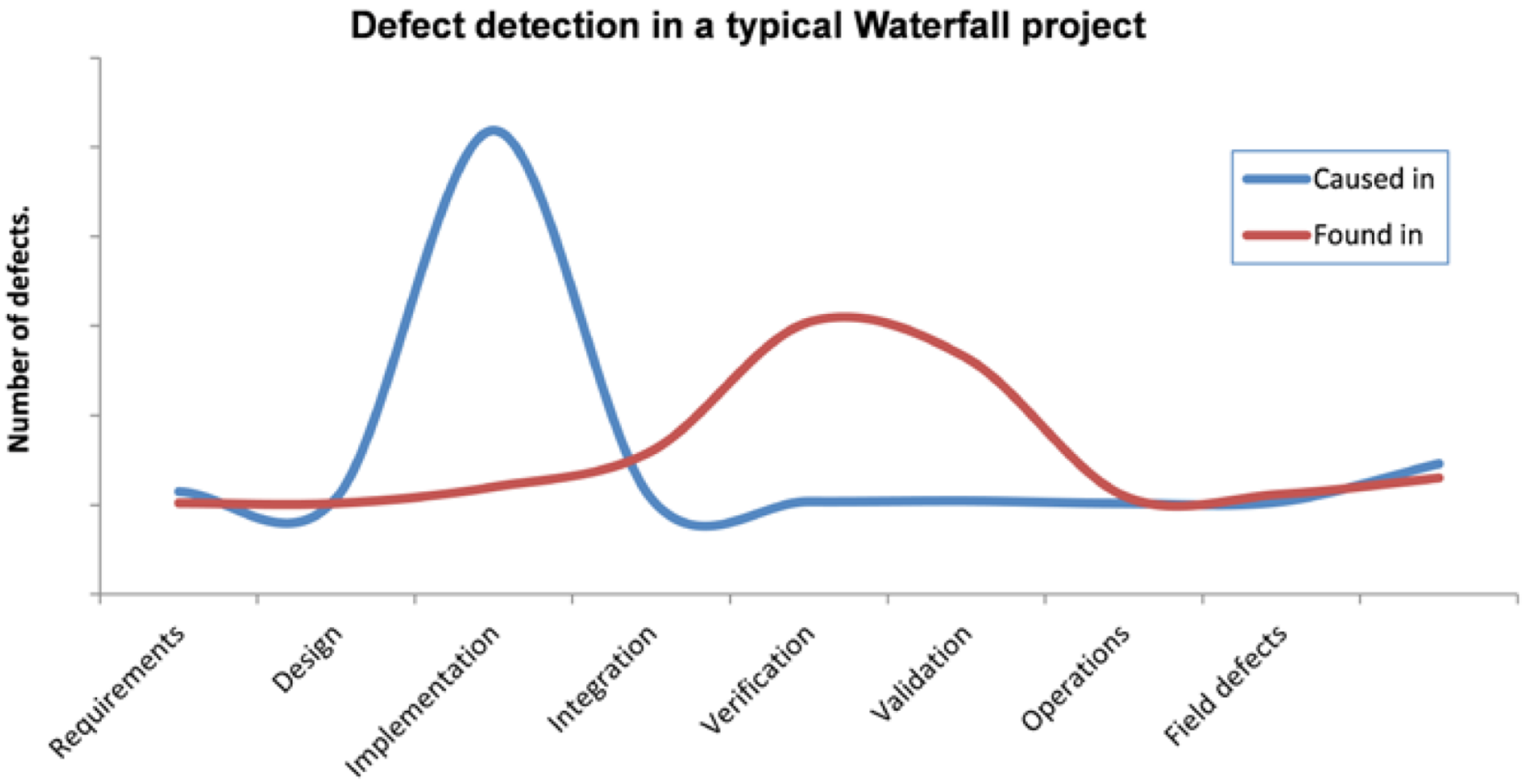
- Questo programma stampa “Hello World”
- The International Obfuscated C Code Contest
  www.ioccc.org/
- Questo è un esempio di codice “altrui” che potremmo dover modificare

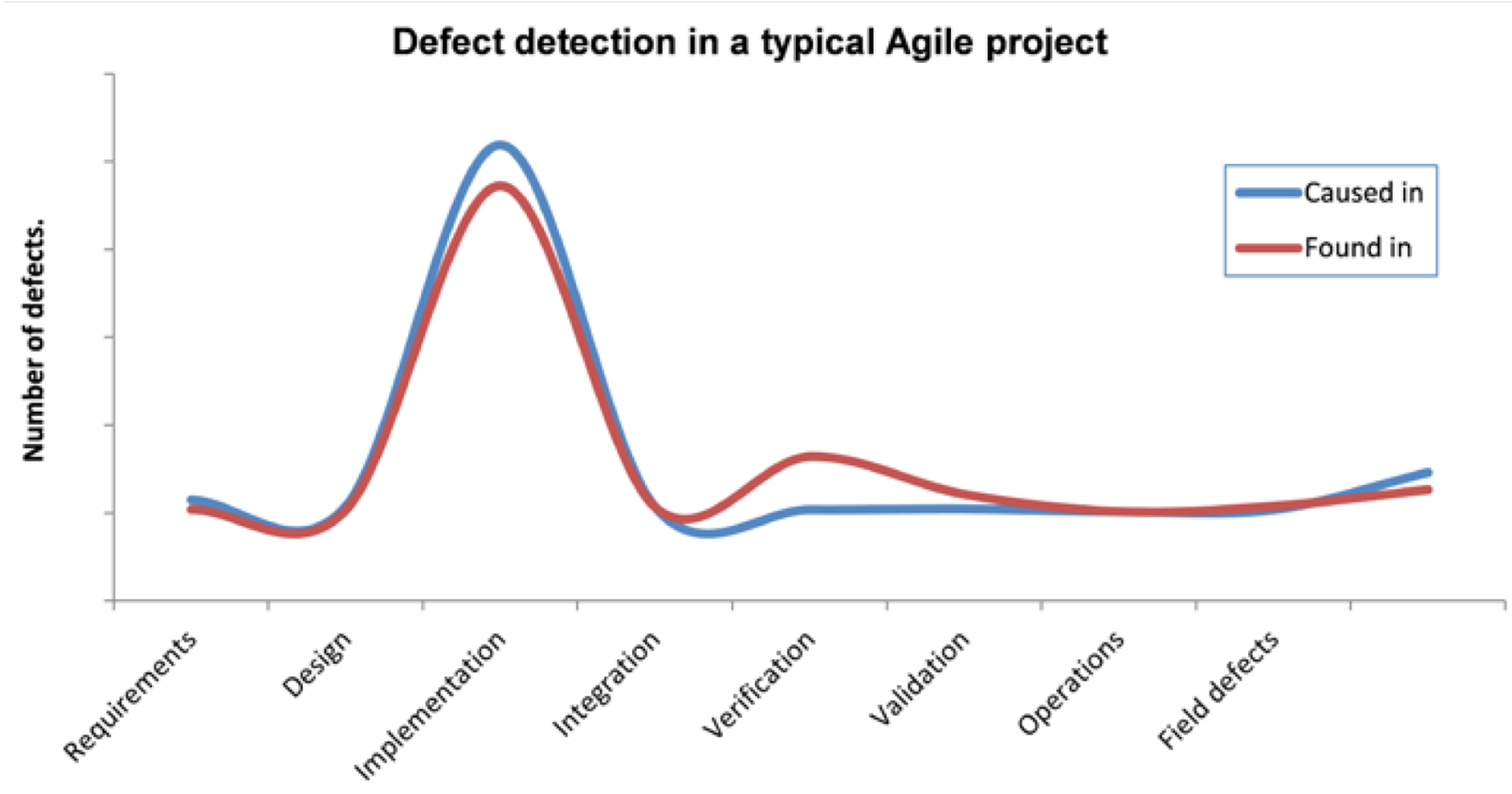
Visualizzare l’evoluzione del sw



Visualizzare l’evoluzione del sw

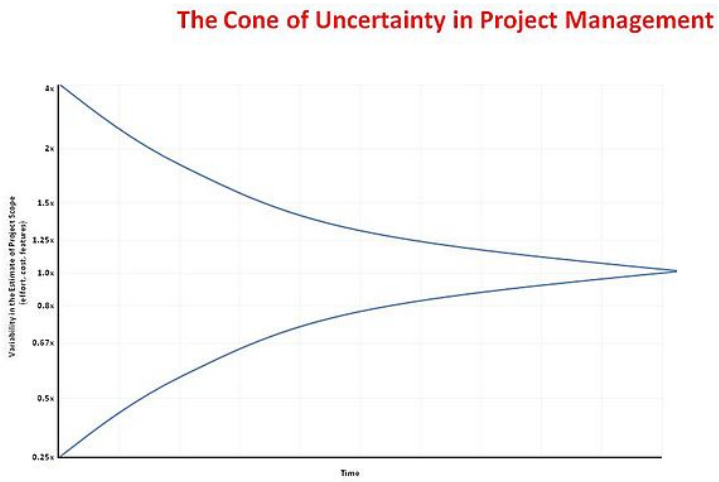
- vis.cs.ucdavis.edu/~ogawa/codeswarm/
- vis.cs.ucdavis.edu/~ogawa/storylines/apache.svg
- code.google.com/archive/p/codeswarm/


Defect detection in a typical Waterfall project
Number of defects.
Caused in
Found in
Requirements
Design
Implementation
Integration
Verification
Validation
Operations
Field defects

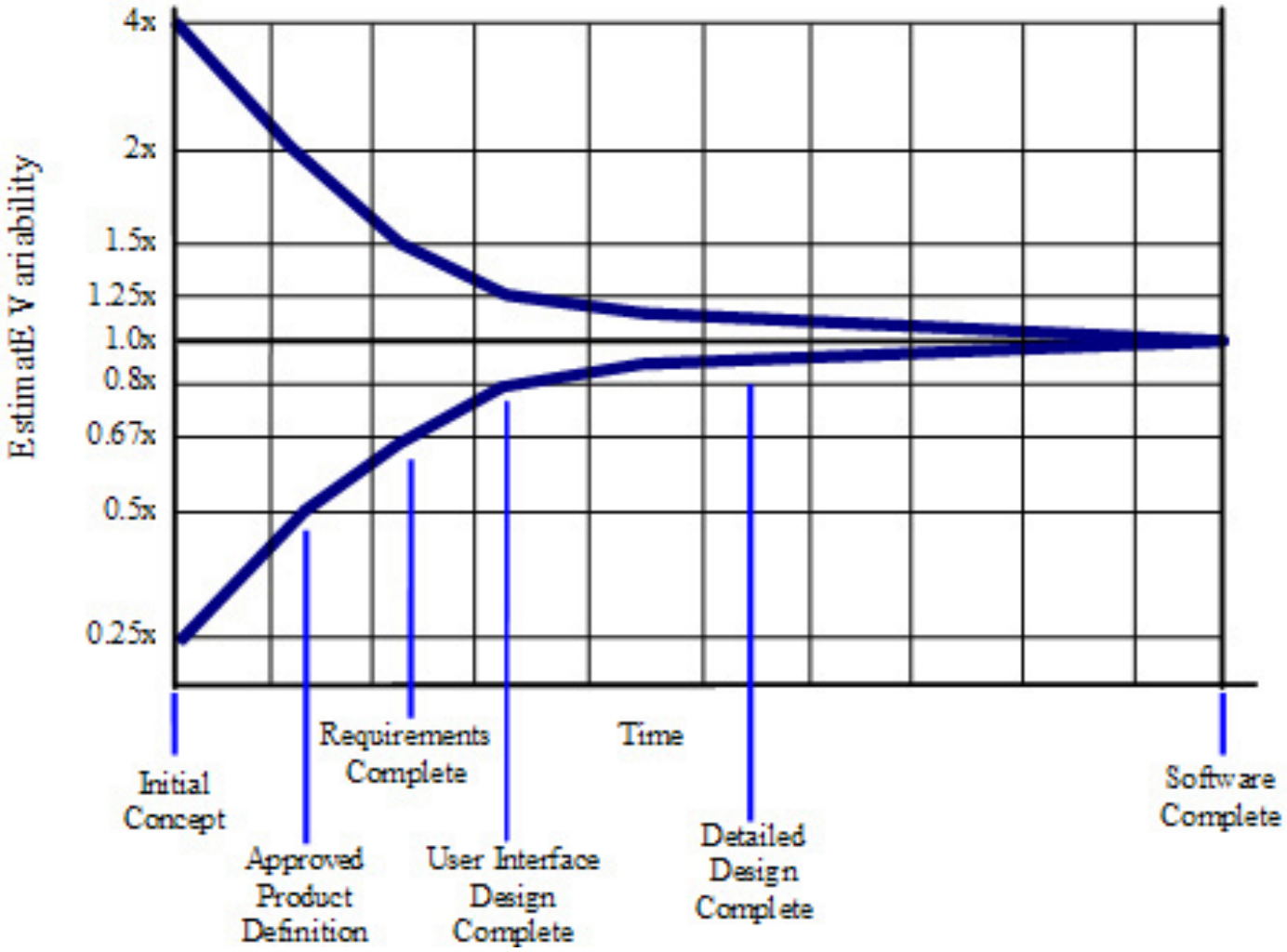

Defect detection in a typical Agile project
Number of defects.
Caused in
Found in
Requirements
Design
Implementation
Integration
Verification
Validation
Operations
Field defects

Alcune leggi empiriche sul sw

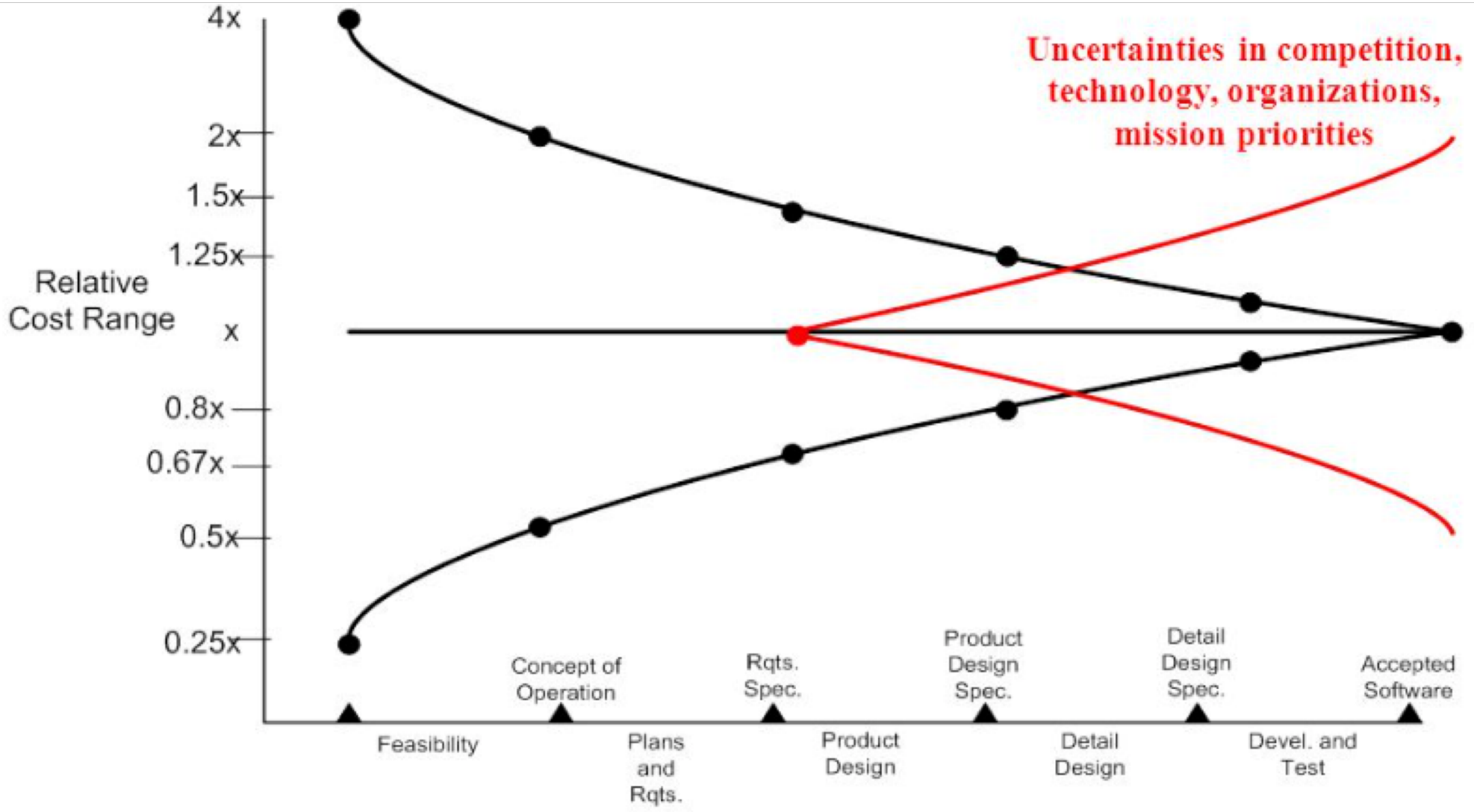
- **La legge di Humphrey**: i requisiti di un nuovo sistema non saranno chiari solo dopo che gli utenti avranno iniziato a usarlo
- **Il lemma di Wegner**: non è possibile specificare completamente un sistema interattivo
- **La legge di Ziv**: l'incertezza è inevitabile quando si produce software; per esempio lo sforzo a consuntivo può differire anche di 16 volte da quello preventivato



Il cono di incertezza dello sviluppo



Il cono di incertezza del ciclo di vita



Evoluzione del sw: definizioni

- *Modifiche fatte ad un programma per computer dopo che è stato consegnato al cliente* (1983)
- *Modifiche ad un prodotto software dopo il rilascio per correggere difetti, migliorare le prestazioni o altre qualità, o per adattarlo ad un diverso ambiente* (1993)
- *Invece di pianificare lo sviluppo, pianificate l'evoluzione del software* (2015)

Evoluzione del sw: sinonimi

- Reingegnerizzazione
- Manutenzione
- Aggiornamento
- Adattamento
- Refactoring
- Trasformazione architetturale

L’evoluzione del software

- La necessità di cambiamenti prima (durante lo sviluppo) e dopo il deployment è l’unico fattore costante nei diversi cicli di vita del sw
- I prodotti software che hanno successo sono quelli che si adattano ai continui mutamenti dei requisiti richiesti dagli stakeholders
- Notabene: Progettare l’evoluzione del software significa spesso progettare l’evoluzione di una famiglia di prodotti o sistemi, invece che un unico sistema

Famiglie di prodotti software (software product lines)

- Famiglie di prodotti o sistemi software basati su una parte comune (di solito preponderante) e una parte variabile
- La gestione di *software product lines* necessita di specifici processi di sviluppo orientati al riuso e alla manutenzione preventiva

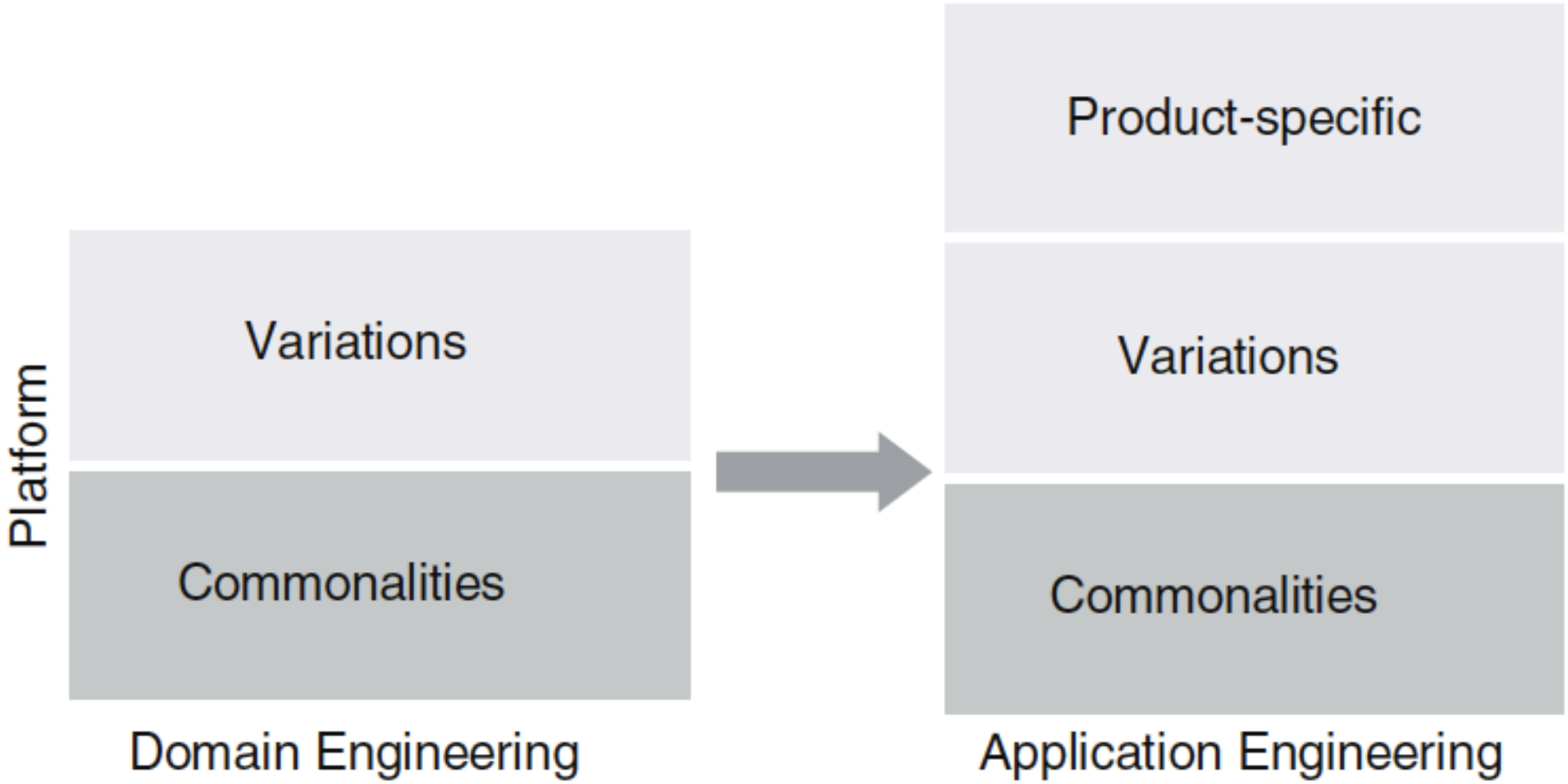


Fig. 1.3. The relation of different types of variability

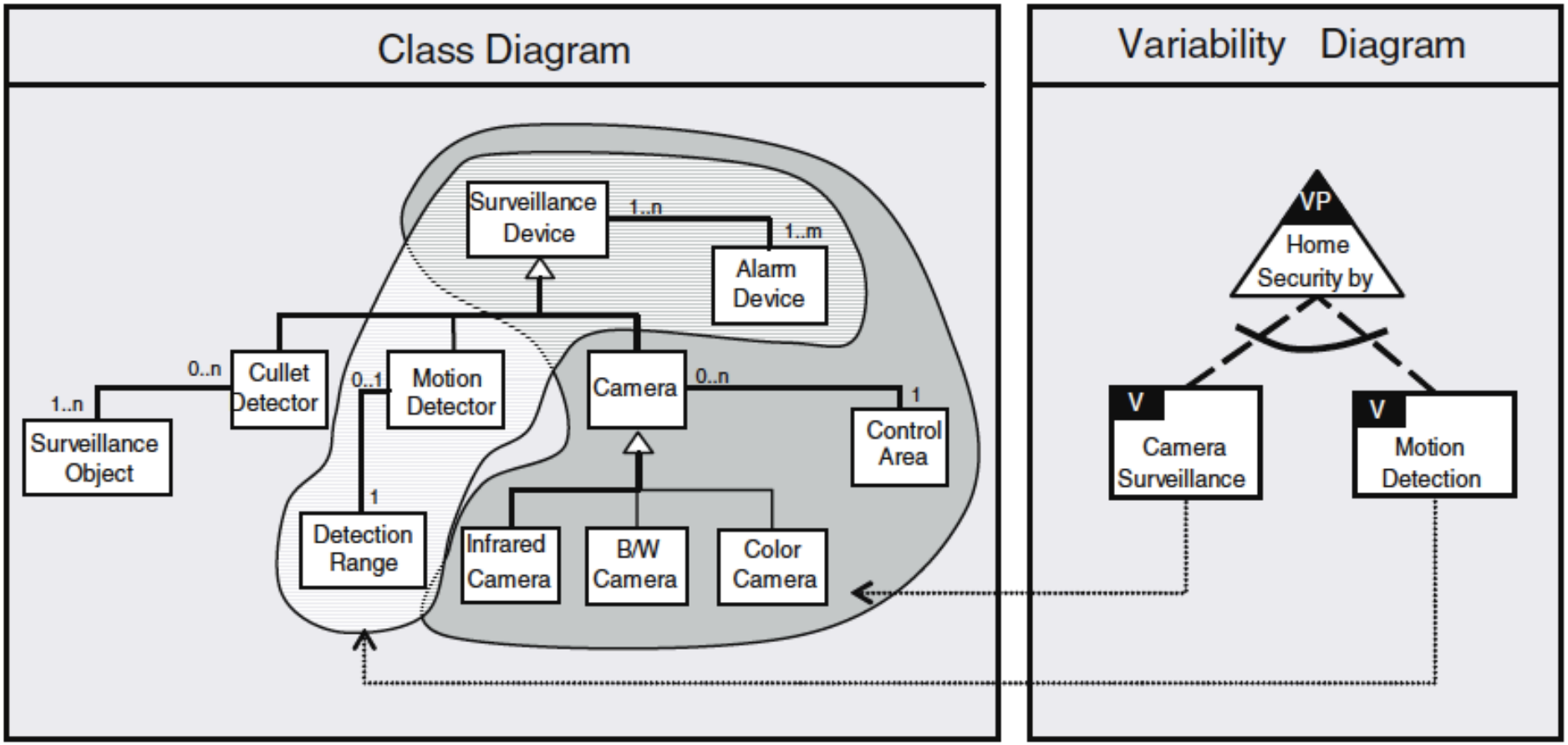


Fig. 1.5. Relation between variability model and class model

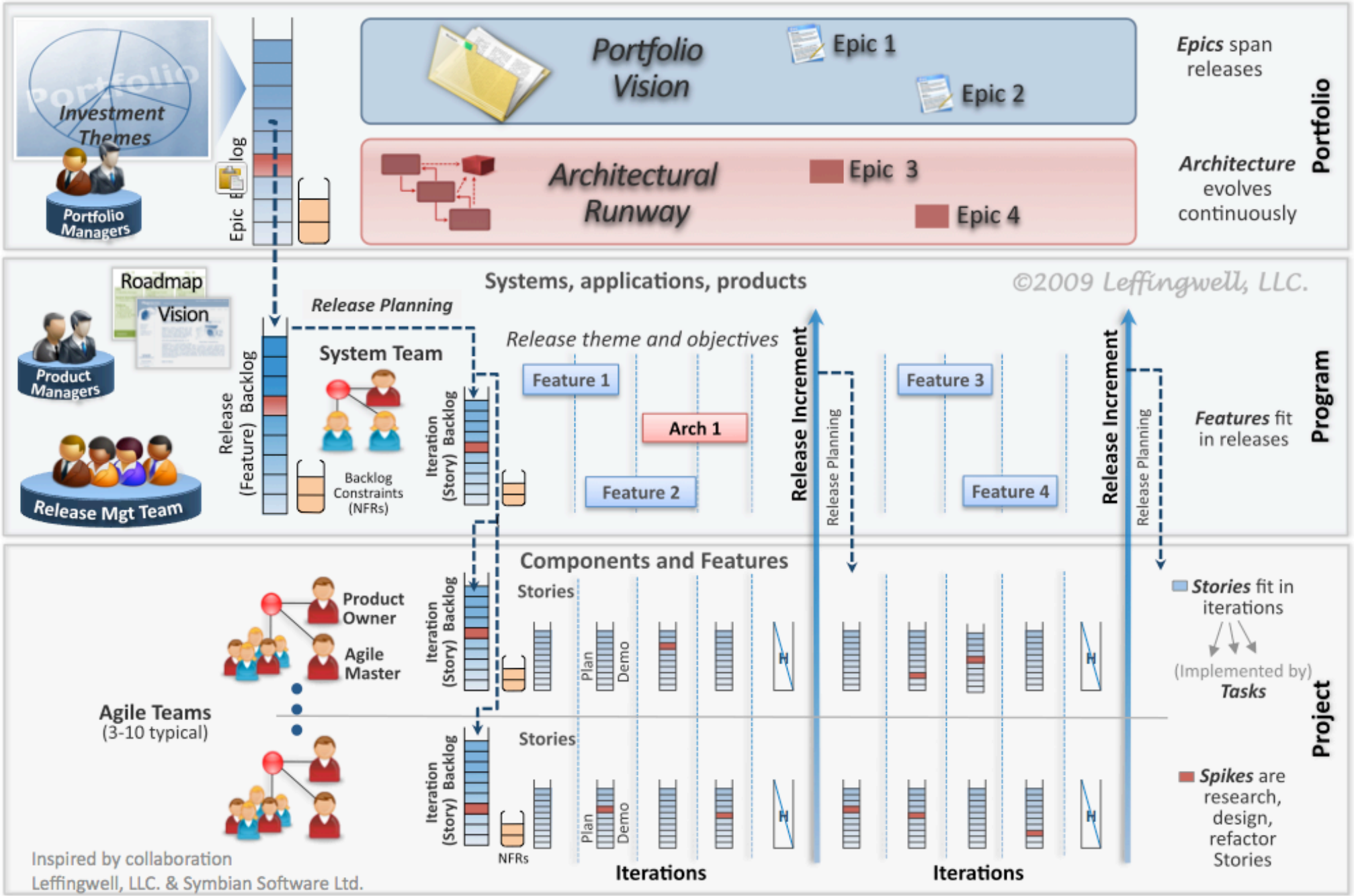
Come evolve un prodotto sw: i rilasci

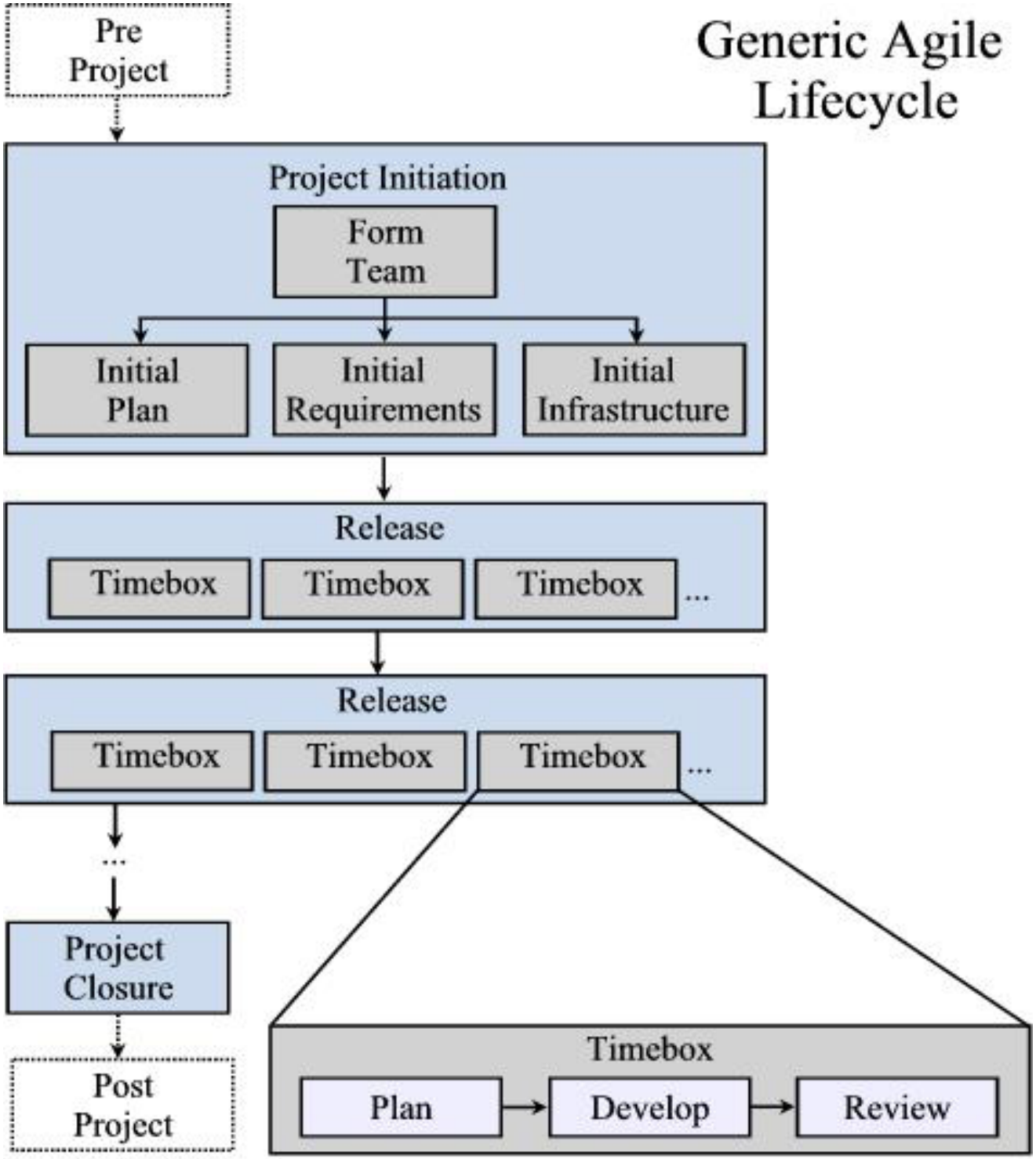
I prodotti software esistono in genere in più versioni di sviluppo, dette **rilasci** (*release*, in inglese)

- *Rilascio corrente*: contiene miglioramenti e aggiustamenti incrementali, in base ai rapporti degli utenti che segnalano difetti
- *Rilascio in campo* (*fielded*): contiene riparazioni urgenti fatte presso l'utente per mantenere operativo il sistema software
- *Rilascio su richiesta* (es. *prodotto in versione alfa*): prodotto che contiene nuove funzioni da testare e convalidare prima di inserirle nel rilascio corrente
- *Rilascio in itinere* (es. *prodotto in versione beta*): prodotto completo ma non ancora testato del tutto, e non necessariamente da distribuire a tutti gli utenti

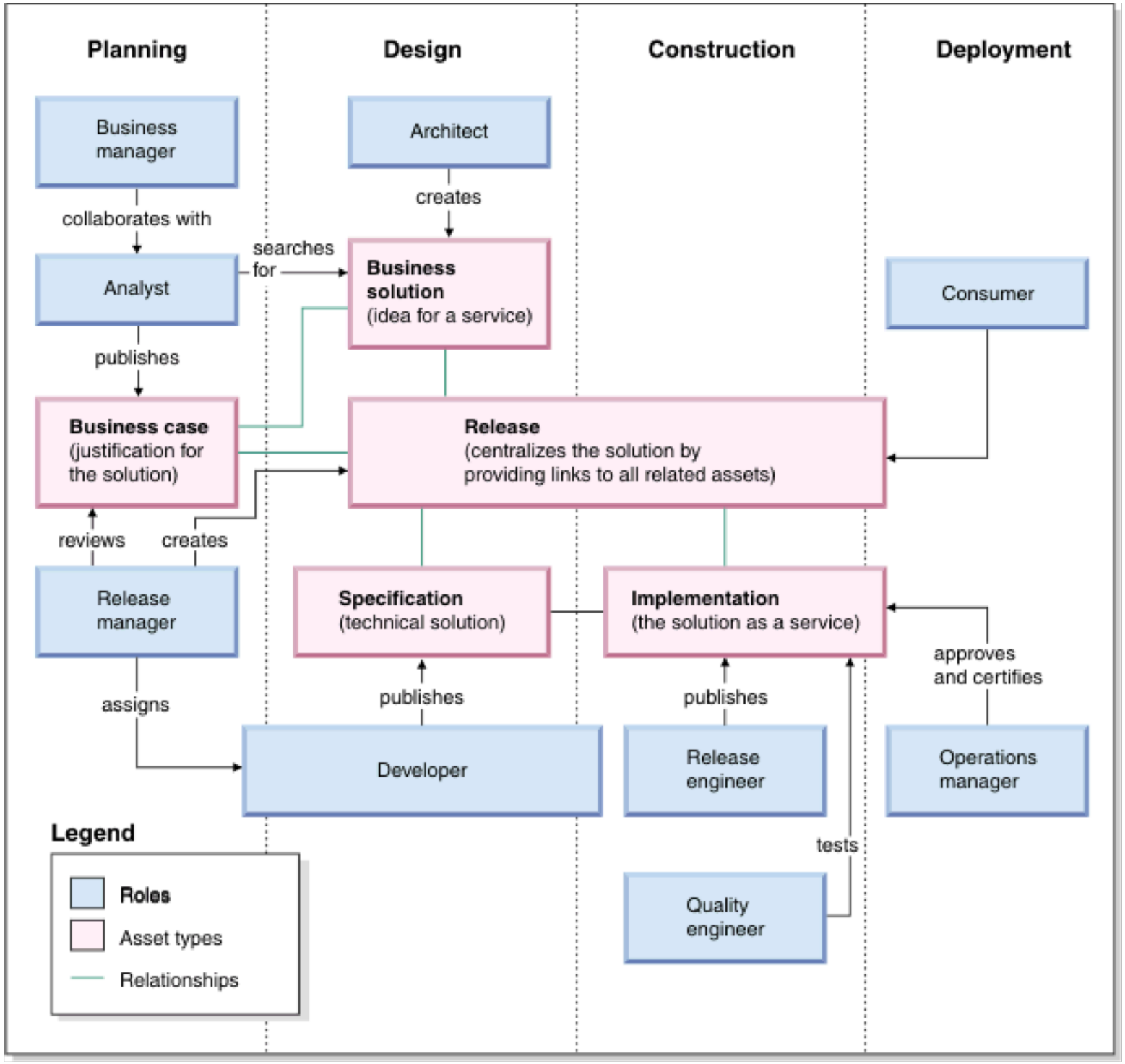
The Agile Enterprise Big Picture

For discussion, see www.scalingsoftwareagility.wordpress.com



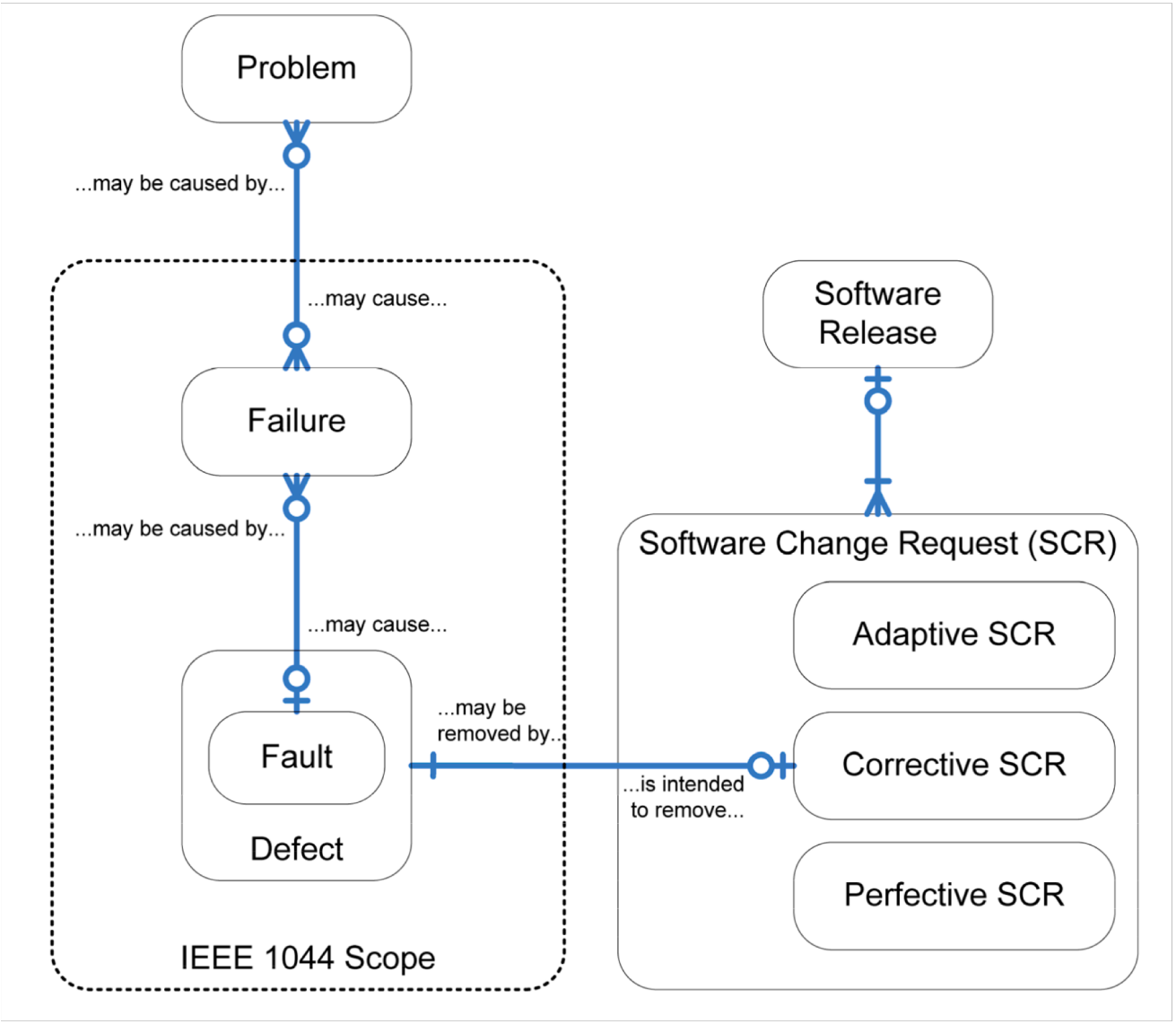

Generic Agile Lifecycle
Pre Project
Project Initiation
Form Team
Initial Plan
Initial Requirements
Initial Infrastructure
Release
Timebox
Timebox
Timebox
...
Release
Timebox
Timebox
Timebox
...
...
Project Closure
Post Project
Timebox
Plan
Develop
Review
© 2008 Steven Thomas

Fasi e ruoli del ciclo di sviluppo

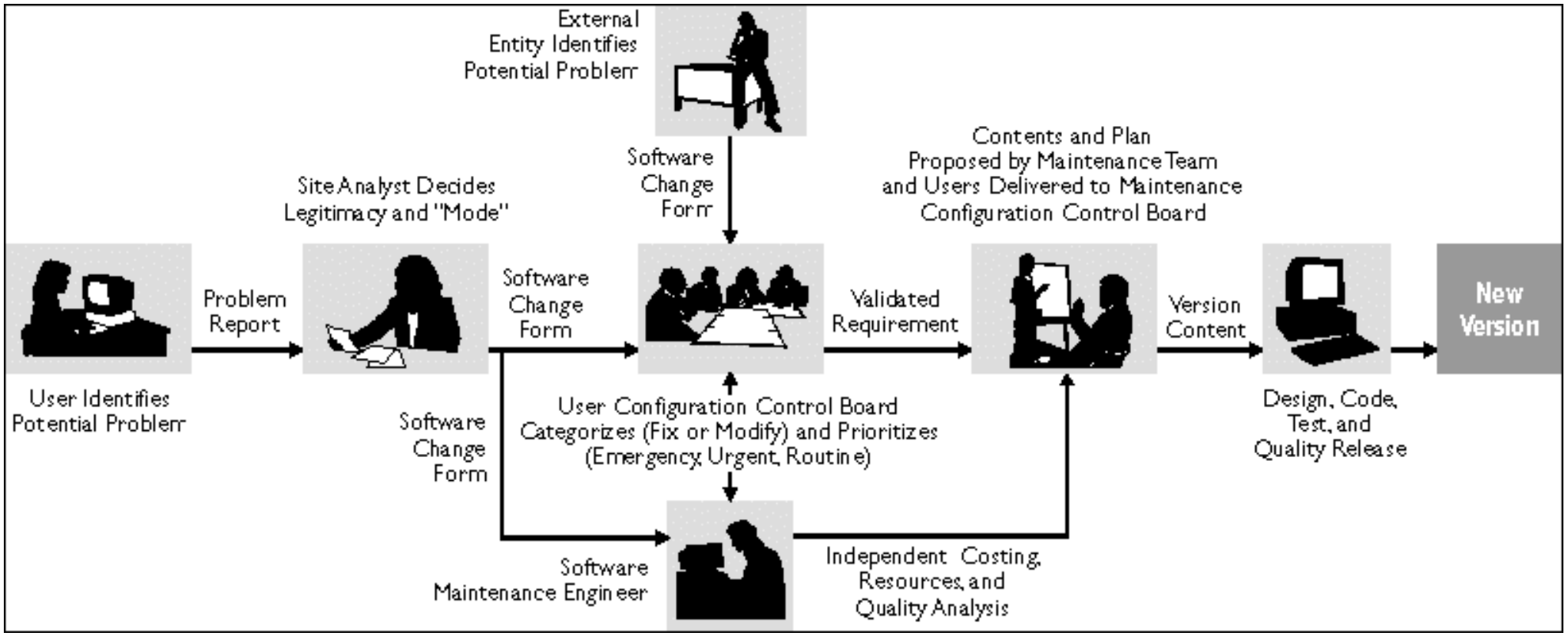


pic.dhe.ibm.com/infocenter/rmchelp/v7r5m2/index.jsp?topic=%2Fcom.ibm.ram.doc%2Ftopics%2Fc_delivery_of_release.html

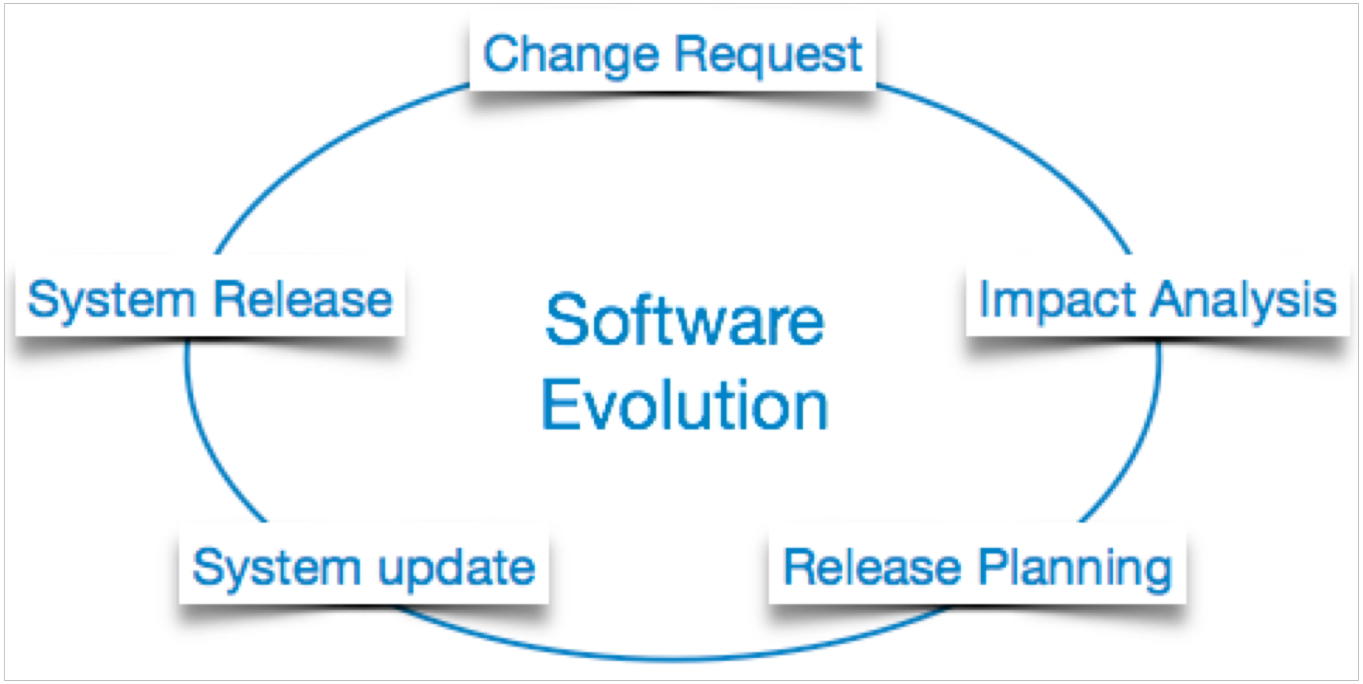
IEEE 1044:
Lo standard terminologico delle anomalie sw



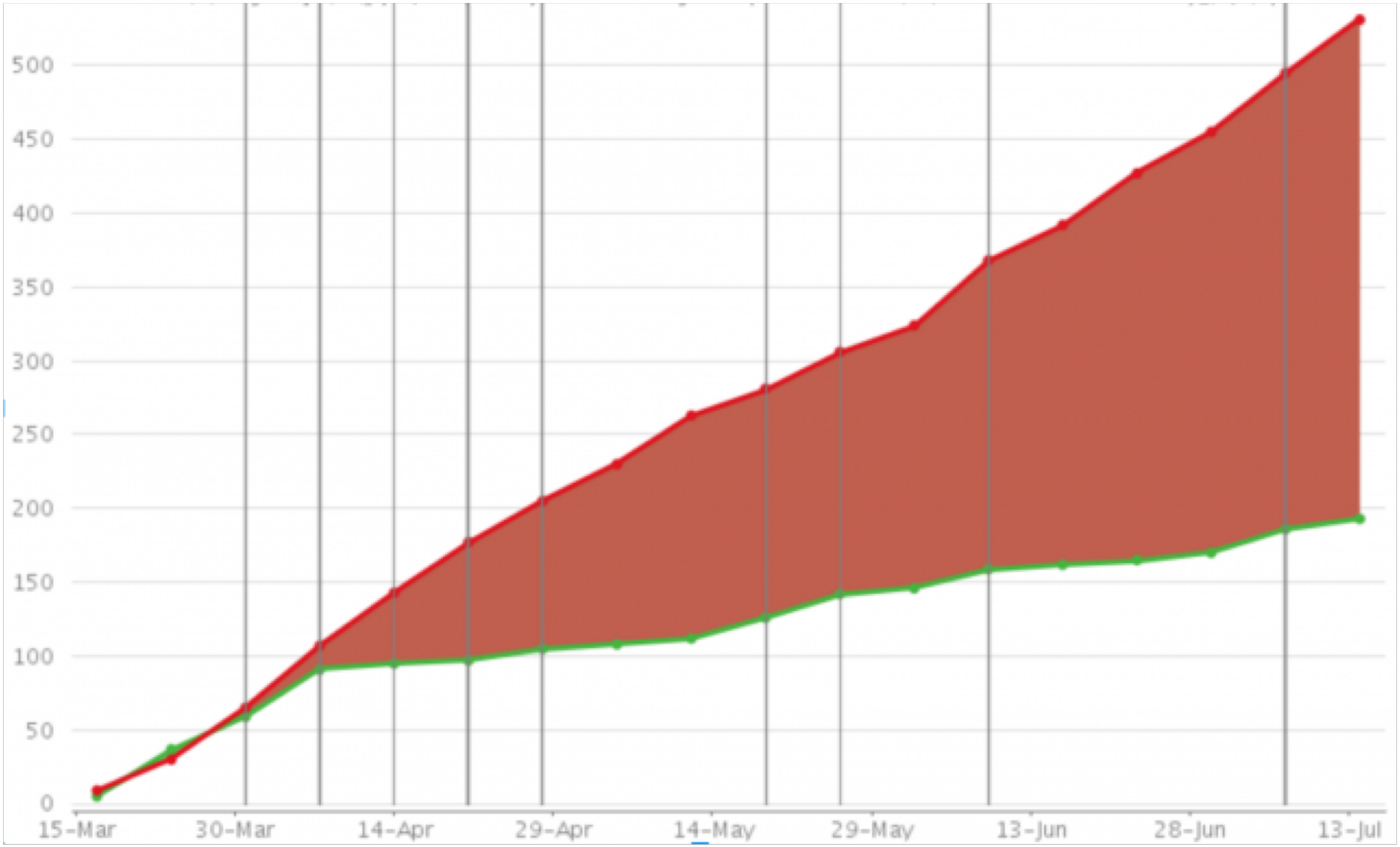
Ciclo di manutenzione



Ciclo di manutenzione: fasi



Strumenti di gestione delle segnalazioni



Linea rossa: segnalazioni aperte; linea verde: segnalazioni chiuse

http://tinyletter.com/programming-beyond-practices/letters/the-sad-graph-of-software-death

IEEE/EIA 12207
Scopo della manutenzione sw

Processo di manutenzione del sw

- Questo processo si attiva quando il prodotto software viene modificato nel codice e nella sua documentazione a causa di un problema o per necessità di miglioramento o adattamento
- ***L'obiettivo del processo è di modificare il prodotto sw preservando la sua integrità***
  - Questo processo include la migrazione o il ritiro del prodotto
  - Il processo di manutenzione termina col ritiro del prodotto

Fonte: IEEE/EIA 12207.0-1996, © IEEE.1998

Persone impiegate per sviluppo o manutenzione sw (solo USA, stime CaperJones)

Year	Development Personnel	Maintenance Personnel	Total Personnel	Maintenance Percent
1950	1,000	100	1,100	9.09%
1955	2,500	250	2,750	9.09%
1960	20,000	2,000	22,000	9.09%
1965	50,000	10,000	60,000	16.67%
1970	125,000	25,000	150,000	16.67%
1975	350,000	75,000	425,000	17.65%
1980	600,000	300,000	900,000	33.33%
1985	750,000	500,000	1,250,000	40.00%
1990	900,000	800,000	1,700,000	47.06%
1995	1,000,000	1,100,000	2,100,000	52.38%
2000	750,000	2,000,000	2,750,000	72.73%
2005	775,000	2,500,000	3,275,000	76.34%
2010	800,000	3,000,000	3,800,000	78.95%
2015	1,000,000	3,500,000	4,500,000	77.78%
2020	1,100,000	3,750,000	4,850,000	77.32%
2025	1,250,000	4,250,000	5,500,000	77.27%

http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf

Sistemi legacy

- I sistemi sw sviluppati su commessa per un'organizzazione spesso hanno una vita operativa lunga
- Molti sistemi sw in uso vennero sviluppati molto tempo fa usando tecnologie oggi obsolete
- Tali sistemi sono spesso **critici**, ovvero essenziali per il funzionamento normale dell'organizzazione che li usa
- Tali sistemi vengono detti **legacy systems** (ereditati)

Definizione: sistema legacy

“Un sistema o prodotto informatico che continua ad essere usato a causa del costo proibitivo di rimpiazzarlo o riprogettarlo, malgrado la sua scarsa competitività e compatibilità con altri sistemi moderni. I sistemi legacy sono di solito grandi, monolitici e difficili da modificare.”

[Free Online Dictionary of Computing]

Sostituire un sistema legacy

- Chi modifica un sistema legacy (di solito per manutenzione *adattiva*) si assume vari **rischi**
  - I sistemi legacy raramente hanno una specifica completa; durante la loro vita hanno subito modifiche anche importanti, di solito non documentate
  - I processi aziendali si basano sul sistema legacy, se lo rompiamo l'azienda si ferma
  - Il sistema legacy può seguire regole aziendali non documentate
  - Lo sviluppo di nuovo sw è rischioso di per sé

Leggi di Lehman

Manny Lehman

- **Legge del cambiamento continuo**:
- un sistema deve evolvere e cambiare per mantenere la sua utilità, perché cambia l'ambiente in cui opera

Leggi di Lehman

Manny Lehman

- **Legge della complessità crescente**:
- la struttura di un sistema che evolve si deteriora man mano che cambia, e devono essere spese risorse aggiuntive per preservarne la funzione e semplificarne la struttura

Leggi di Lehman

- **Legge dell'evoluzione dei grandi programmi**:
- l'evoluzione di un programma grande è un processo "*autoregolante*". Sono all'incirca invarianti per ciascuna versione attributi critici quali la dimensione, il periodo di tempo necessario per la versione successiva, il numero di errori rilevati

Leggi di Lehman

- **Legge di stabilità organizzativa**:
- durante il ciclo di vita di un programma la sua velocità di sviluppo è all'incirca costante e indipendente dalle risorse dedicate allo sviluppo del sistema stesso

Leggi di Lehman

- **Legge di conservazione della familiarità**:
- durante il ciclo di vita di un sistema il tasso di cambiamento di ciascuna versione è approssimativamente costante

Le leggi di Lehman

Support	Brief Name	Description
I 1974	Continuing Change	An *E*-type system must be **continually adapted** else it becomes progressively **less satisfactory** in use
II 1974	Increasing Complexity	As an *E*-type system is evolved its **complexity increases** unless **work is done** to maintain or reduce it
III 1974	Self Regulation	Global *E*-type system evolution processes are **self-regulating**
IV 1978	Conservation of Organisational Stability	**Average activity rate** in an *E*-type process tends to **remain constant** over system lifetime or segments of that lifetime
V 1978	Conservation of Familiarity	In general, the **average incremental growth** (growth rate trend) of *E*-type systems tends to **decline**
VI 1991	Continuing Growth	The **functional capability** of *E*-type systems must be continually enhanced to **maintain user satisfaction** over system lifetime
VII 1996	Declining Quality	**Unless rigorously adapted** to take into account changes in the operational environment, the quality of an *E*-type system will appear to be **declining** as it is **evolved**
VIII 1996	Feedback System (Recognised 1971, formulated 1996)	*E*-type evolution processes are multi-level, not stricly hierarchical, multi-loop, multi-agent **feedback systems**

Le leggi dell'evoluzione del sw (sommario)

Il software che rimane utile nel tempo cambierà di continuo:

- Diventerà più complesso
- Aumenterà la sua dimensione
- Diminuirà la sua qualità

Tipi di manutenzione del sw

- Modificare un prodotto sw dopo il deployment è normale
  - Durante l'uso emergono nuovi requisiti (m. perfettiva)
  - L'ambiente operativo del sw cambia oppure viene aggiunto nuovo hardware oppure occorre migliorare prestazioni e affidabilità (m. adattiva)
  - Gli errori emersi durante l'uso vanno riparati (m. correttiva)
- Tutte le organizzazioni hanno il problema di gestire i propri sistemi legacy, cioè i sistemi software la cui utilità persiste nel tempo - anche decenni!
- La gestione di famiglie di prodotti software (Software Product Lines) anch'essa necessita di tecniche efficaci per gestirne l'evoluzione

Tipi di manutenzione del sw

- Manutenzione correttiva
- Manutenzione perfettiva
- Manutenzione adattiva
- Manutenzione preventiva
- Manutenzione d'emergenza

Correggere gli errori

- Manutenzione **preventiva**
  - Identifica e rimuove errori latenti, prima che vengano riscontrati
  - Adatta a sistemi con problemi di sicurezza
- Manutenzione **correttiva**
  - Identifica e rimuove i difetti riscontrati durante l'uso
  - Corregge gli errori residui dopo il testing pre-delivery
- Manutenzione **d'emergenza**
  - Manutenzione correttiva imprevedibile e urgente
  - Rischiosa perché prevede testing ridotto

Manutenzione di sviluppo

- Manutenzione **perfettiva**
  - Migliora le prestazioni, l'affidabilità, la manutenibilità
  - Aggiunge nuove funzionalità su richiesta del cliente
- Manutenzione **adattiva**
  - Sviluppo o migrazione
  - Adatta ad un nuovo ambiente (hw, sistema operativo, middleware)

Manutenzione preventiva

- Le famiglie di prodotti software (SPL) pongono speciali problemi di manutenzione
- Quando un cliente segnala un problema che viene corretto (manutenzione *correttiva*), tutte le versioni rilasciate dello stesso prodotto dovrebbero essere corrette: questa è la manutenzione *preventiva*

Il costo della manutenzione

L'incidenza della manutenzione sul costo complessivo di sviluppo di un sistema è crescente nel tempo (seconda Legge di Lehman) e può giungere sino all'80%, di cui

- Manutenzione correttiva: 20%
- Manutenzione adattiva: 25%
- Manutenzione **perfettiva**: 55%

La maggioranza dei costi è attribuibile ai mutamenti imposti dall'ambiente (1° Legge di Lehman)

Fattori di costo della manutenzione

- Instabilità del personale
  - I costi di manutenzione aumentano se il personale addetto cambia spesso
- Irresponsabilità dei progettisti
  - Se gli sviluppatori del sistema non hanno responsabilità di manutenzione non c'è incentivo per "*design for change*"
- Inesperienza del personale
  - Il personale addetto alla manutenzione è spesso poco esperto e con scarsa conoscenza del dominio applicativo
- Età e struttura del software
  - Quando il programma invecchia la sua struttura si degrada e diventa più difficile da capire e modificare

Discussione

Come possiamo definire i costi di manutenzione?

Stimare il costo di manutenzione

- La **predizione dei costi** di manutenzione di un sistema software si basa sulla valutazione di complessità dei componenti del sistema
- Esiste evidenza empirica che gran parte dello sforzo di manutenzione di solito si spende su un piccolo numero di componenti
- La complessità di un componente sw dipende da
  - Strutture di controllo
  - Strutture dati
  - Dimensioni di procedure e moduli

Metriche di processo

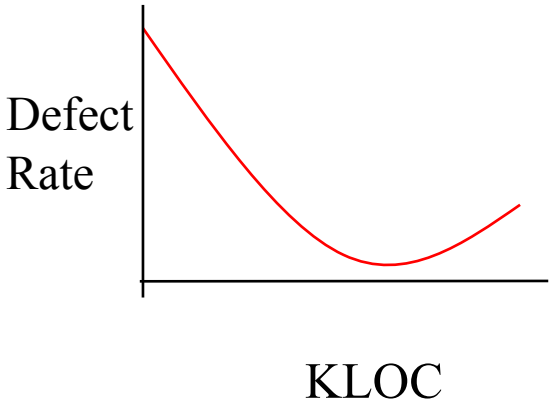
- Alcuni **indicatori di processo** permettono di stimare i costi di manutenzione
  - # richieste di manutenzione correttiva
  - Tempo medio necessario per analisi d'impatto di una modifica
  - Tempo medio necessario per soddisfare una richiesta di modifica
  - # richieste di modifica principale
- Se i valori di queste misure aumentano, abbiamo un'indicazione di manutenibilità decrescente e conseguente aumento dei costi relativi

Metriche di prodotto

- **Dimensione**: Lines of Code (LOC o KLOC)
- **Complessità ciclomatica** di McCabe: misura il numero di cammini indipendenti nel flowchart del programma (o il numero di condizioni binarie)
- **Complessità di Halstead** (o *Software Science*): misura il numero di operatori e operandi nel programma
- **Metriche strutturali**: misure di accoppiamento basate su Fan in e Fan out dei moduli

Uso delle LOC nella manutenzione

- La dimensione del programma in KLOC influenza il costo di manutenzione: programmi più grandi richiedono un maggiore sforzo di manutenzione
- Il numero di difetti per KLOC (Tasso di difettosità, o *Defect Rate*) ha una relazione con la dimensione dei moduli
- Esiste una dimensione ottima dei moduli, che induce il minimo tasso di difettosità; tale dimensione però dipende dal prodotto che si sviluppa, e dal linguaggio e ambiente operativo usati



Complessità ciclomatica di McCabe

- La **complessità ciclomatica di McCabe** è una delle più note metriche di manutenzione [McCabe 76]
- Stima la manutenibilità del programma (testabilità e comprensibilità) misurando il numero di cammini esecutivi indipendenti; tale numero è un indicatore dello sforzo necessario per testare un programma
- La complessità ciclomatica cerca di valutare quanta "fiducia" possiamo avere che un programma sia privo di errori
- Viene usata spesso in combinazione con altre metriche
- È indipendente dal linguaggio di programmazione
- È stata estesa per comprendere la complessità progettuale di un sistema [McCabe 89]

Complessità ciclomatica (M)

M si calcola sul diagramma di flusso del programma

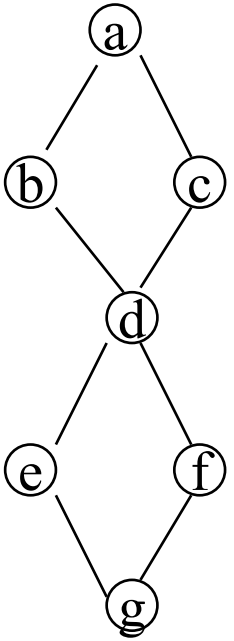
$$M = e - n + 2p$$

dove

- e = Numero di archi
- n = Numero di nodi
- p = Numero di sottografi sconnessi

Esempio 1

- Programma con due comandi condizionali (if-then-else)
  - 8 archi
  - 7 nodi
  - 1 solo sottografo
  - *M* = 8 – 7 + 2 * 1 = 3
  - *M* = Numero di condizionali + 1
- Un condizionale a n vie (`case select`) viene contato come n-1 condizionali binari
- Il test di iterazione in un comando iterativo vale come un condizionale binario

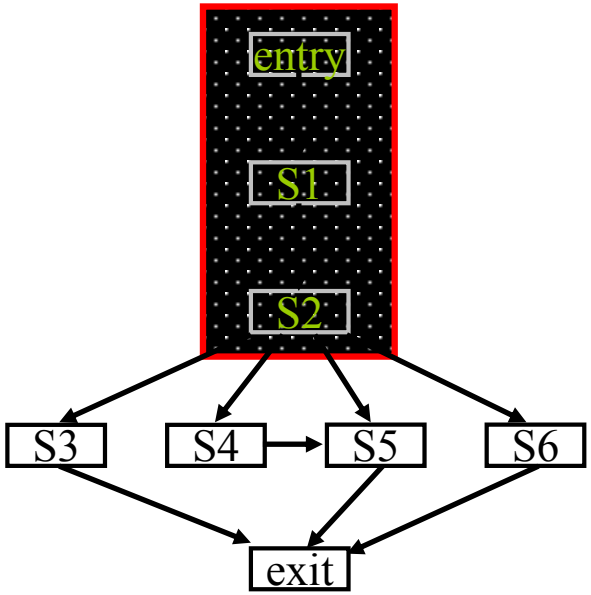


Esempio 2

- e = 8
- n = 6
- p = 1
- M = 4
- M= 4-1+1

(**case** a 4 vie -1) + 1

```
Procedure Trivial
S1 read (n)
S2 switch (n)
     case 1:
S3     write ("one")
       break
     case 2:
S4     write ("two")
     case 3:
S5     write ("three")
       break
     default
S6     write ("Other")
   endswitch
end Trivial
```



Complessità ciclomatica

- McCabe misurò parecchi programmi, stabilendo intervalli di complessità che aiutano il progettista a determinare il rischio e la stabilità del suo programma
- Questa misura si può usare durante lo sviluppo, la manutenzione e la reingegnerizzazione per ottenere stime di rischio, costo o stabilità del programma
- Studi empirici mostrano una buona correlazione tra la complessità ciclomatica di un programma e la frequenza di errori
- Una bassa complessità ciclomatica è indicatrice di buona comprensibilità del programma e sua modificabilità a basso rischio

Complessità ciclomatica di un modulo

- La complessità ciclomatica di un modulo è un ottimo indicatore della sua testabilità
- Un'applicazione tipica della complessità ciclomatica di un modulo è la seguente **scala di valori di rischio**

Complessità ciclomatica	**Valutazione del rischio**
1-10	programma semplice, rischio minimo
11-20	rischio moderato
21-50	programma complesso, rischio alto
maggiore di 50	programma non testabile, rischio altissimo

Complessità ciclomatica

- La complessità ciclomatica si può usare per:
  - **Analisi del rischio durante la stesura del codice**
  - **Analisi del rischio di cambiamento durante la manutenzione**. La complessità del codice aumenta nel tempo a causa delle manutenzioni. Misurando la complessità ciclomatica prima e dopo un cambiamento si può monitorare la configurazione del sistema e decidere come mimizzare il rischio di ulteriori cambiamenti
  - **Reingegnerizzazione**. L'analisi della complessità ciclomatica valuta la struttura di un sistema: dunque il rischio di reingegnerizzazione di una parte del codice è correlato alla sua complessità.

Complessità ciclomatica

- La complessità ciclomatica si può usare durante il testing:
  - Definisce il numero di cammini indipendenti da testare
  - *Path coverage set* = insieme dei cammini che eseguiranno tutti i comandi e valuteranno tutte le condizioni logiche almeno una volta
  - Obiettivo: definire un insieme minimale di casi di test
  - Il *Path coverage set* non è unico!

Complessità di Halstead

- La misura della **Complessità di Halstead** si applica direttamente al sorgente del programma
- Si può usare direttamente come metrica di manutenzione
- Tuttavia il suo uso non gode di consenso unanime: le opinioni in merito variano da “oscura e inaffidabile” [Jones 94] a “una delle migliori metriche di manutenibilità” [Oman 91]

Complessità di Halstead

- Un programma si compone di un numero finito di token, classificati in “operatori” e “operandi”
- **Operatore** è ogni simbolo o parola chiave che specifica un’ azione
- **Operando** è ogni simbolo che rappresenta dati
- La complessità di un programma, ed il suo tempo di sviluppo, è proporzionale al tipo e numero di operatori e operandi che contiene

n_1 = # di operatori distinti in un programma

n_2 = # di operandi distinti in un programma

N_1 = # di occorrenze di operatori

N_2 = # di occorenze di operandi

Maurice Halstead

Misure di Halstead

Lunghezza del programma $N = N_1 + N_2$

Vocabolario del programma $n = n_1 + n_2$

Volume $V = N * (log_2\ n)$

Difficoltà $D = (n_1/2) * (N2/n2)$

Sforzo $E = D * V$

Esempio

Procedura FORTRAN che ordina un vettore

```
SUBROUTINE  SORT (X, N)
     INTEGER X(100), N, I, J, SAVE, IM1
C    THIS ROUTINE SORTS ARRAY X INTO ASCENDING ORDER
     IF(N .LT. 2) GO TO 220
     DO 210 I = 2,N
          IM1 = I - 1
          DO 200 J = 1,IM1
               IF(X(I) .GE. X(J)) GO TO 200
                    SAVE = X(I)
                    X(I) = X(J)
                    X(J) = SAVE
200       CONTINUE
210  CONTINUE
220 RETURN
END
```

Esempio

Procedura FORTRAN che ordina un vettore

```
SUBROUTINE  SORT (X, N)
      INTEGER X(100), N, I, J, SAVE, IM1
C     THIS ROUTINE SORTS ARRAY X INTO ASCENDING
C     ORDER
      IF(N .LT. 2) GO TO 220
      DO 210 I = 2,N
            IM1 = I - 1
            DO 200 J = 1,IM1
                  IF(X(I) .GE. X(J)) GO TO 200
                        SAVE = X(I)
                        X(I) = X(J)
                        X(J) = SAVE
200                 CONTINUE
210  CONTINUE
220 RETURN
END
```

Operators	Occurrences	Operands	Occurrences
SUBROUTINE	1	SORT	1
()	10	X	8
,	8	N	4
INTEGER	1	100	1
IF	2	I	6
.LT.	1	J	5
GOTO	2	SAVE	3
DO	2	IM1	3
=	6	2	2
-	1	200	2
.GE.	1	210	2
CONTINUE	2	1	2
RETURN	1	220	3
End-of-line	13		
n1 = 14	N1 = 51	n2 = 13	N2 = 42

Esempio

- Lunghezza del programma: N = N1 + N2 = 93
- Vocabolario del programma: n = n1 + n2 = 27
- Volume del programma: $V = N * \log_2 n$

  $= 93 * \log_2 27 = 442.2045$

  – Il volume rappresenta la dimensione di memoria necessaria per una traduzione in binario del programma originale
  – Stima il numero di “confronti mentali” necessari

n1 = 14

N1 = 51

n2 = 13

N2 = 42

Esempio

n1 = 14 N1 = 51

n2 = 13 N2 = 42

- Difficoltà: D = (n1/2) * (N2/n2)

  = (14/2) * (42/13) = 22.6154

  – La difficoltà aumenta se vengono aggiunti altri operatori (cioè quando aumenta n1) e se un operando viene usato ripetutamente (cioè quando aumenta N2/n2)

- Sforzo: E = D * V = 442 .2045 * 22.6154 = 10000

  – Misura di "confronti mentali elementari"

  – Lo psicologo John Stroud suggerì che la mente umana è capace di eseguire un piccolo numero di confronti elementari per secondo, da 5 a 20: questo si chiama "Numero di Stroud"

  – Usando un Numero di Stroud di 15, si ha che il Tempo di Sviluppo T:

    T = E / 15 secondi = 10000/15 secondi = 667 secondi = 11.1 minuti

Complessità di Halstead: critiche

Ambiguità della definizione di operatore e operando

Può essere difficile decidere cos’è operatore e cos’è operando

Le metriche di Halstead misurano la complessità lessicale (testuale) più che la complessità strutturale (logica)

Le basi teoriche sono deboli

L’analisi empirica è limitata

asetechs.com/NewSite2011/Products/Interpreting_Halstead_metrics.htm#BOTTOM

Metriche di struttura modulare

- Gli indicatori KLOC, Complessità Ciclomatica, Complessità di Halstead misurano la complessità dei singoli moduli come se fossero entità separate
- Le metriche di struttura modulare stimano invece l'accoppiamento tra moduli
- ***Fan-in***: # di moduli che chiamano un certo modulo.
- ***Fan-out***: # di moduli chiamati da un certo modulo.
  - I moduli con grande *fan-in* di solito sono piccoli e apprtengono ai livelli inferiori di un'architettura sw a livelli. I moduli grossi e complessi hanno di solito *fan-in* piccolo. I moduli con grande *fan-out* dipendono da parecchi moduli e alzano il livello di accoppiamento
- Dunque se ci sono moduli con *fan-in* piccolo e *fan-out* grande abbiamo un'indicazione di cattiva qualità di progettazione

Metriche di manutenibilità

- La manutenibilità è la misura in cui un sistema software ammette modifiche per estendere il suo tempo di uso
- Prospettive sulla manuteniblità:
  - **Interna:** Dipende solo dalle caratteristiche del sistema
  - **Esterna:** Dipende anche dal personale di manutenzione, dall'ambiente di supporto, dalla documentazione e dagli strumenti disponibili

Metriche di manutenibilità

Valori bassi sono migliori:

- Quantità di risorse necessarie
- # persone necessarie (manutentori) / dimensione del codice
- Tempo per costruire, eseguire e testare
- Complessità del codice
- Numero di versioni da gestire
- Tempo medio per fare una modifica
- Costo medio per correggere un difetto dopo il rilascio
- # di file oggetto ricompilati dopo una modifica ad un file sorgente
- Quantità di codice morto
- Grado di accoppiamento tra i moduli

Metriche di manutenibilità

Valori alti sono migliori:

- Numero di design pattern usati
- Livello del linguaggio di programmazione
- Rapporto tra linee commento e linee codice
- Numero di test di regressione
- Percentuale dei manutentori che dicono che il codice è facilmente modificabile
- Rapporto tra numero di moduli e dimensione del codice
- Numero di moduli ad alta coesione sul totale dei moduli

Indice di manutenibilità

- Nel 1991 Oman e Hagemeister introdussero una metrica composita per quantificare la manutenibilità del software
- Questo **Indice di Manutenibilità** (Maintainability Index o **MI**) ha oggi parecchie varianti ed è stato applicato con successo a molti sistemi sw industriali
- MI incorpora le metriche di Halstead (sforzo o volume), la complessità ciclomatica di McCabe, la dimensione in LOC, ed il conteggio dei commenti
- `www.stsc.hill.af.mil/crosstalk/2001/08/welker.html`

Indice di manutenibilità (MI)

MI = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g')
- 16.2 * ln (aveLOC) + 50 * sin (sqrt(2.4 * perCM))

- I coefficienti ponderali sono stati derivati empiricamente
- I termini dell'equazione sono:
  - aveV = Volume di Halstead medio V per modulo
  - aveV(g') = complessità ciclomatica estesa media per modulo
  - aveLOC = media LOC per modulo; e, *opzionalmente*
  - perCM = percentuale media di righe commento per modulo

Indice di manutenibilità

Un'altra versione omette il termine *"perCM"*

MI = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC)

Questa composizione di metriche si spiega perché si voleva incorporare nell'indice complessivo:

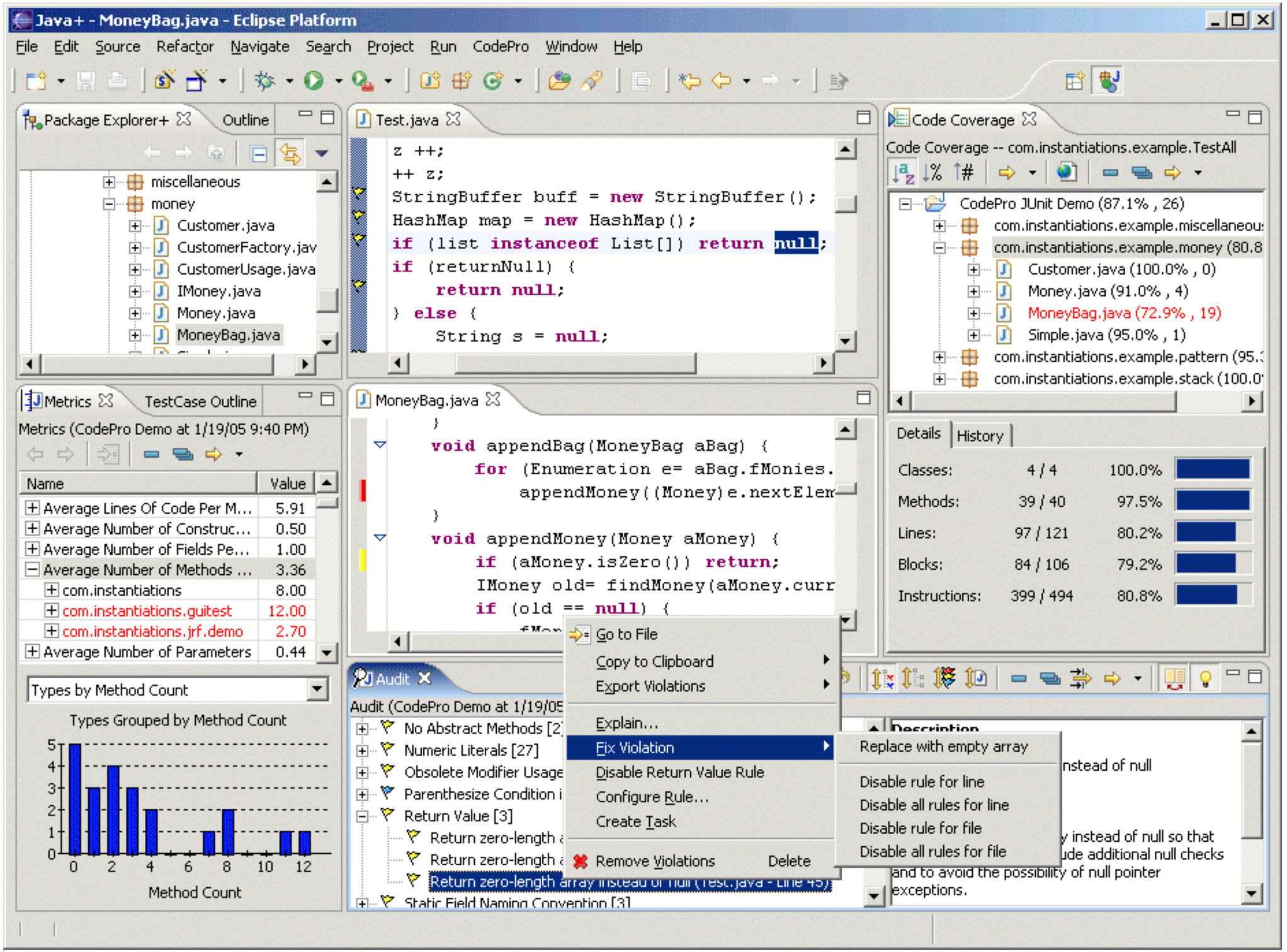
- La densità di operatori e operandi (quante variabili ci sono e come vengono usate)
- La complessità logica (da quanti cammini esecutivi è composto il codice)
- La dimensione (quanto codice c'è)
- L'apporto dell'intuizione umana (commenti nel codice).

Indice di manutenibilità

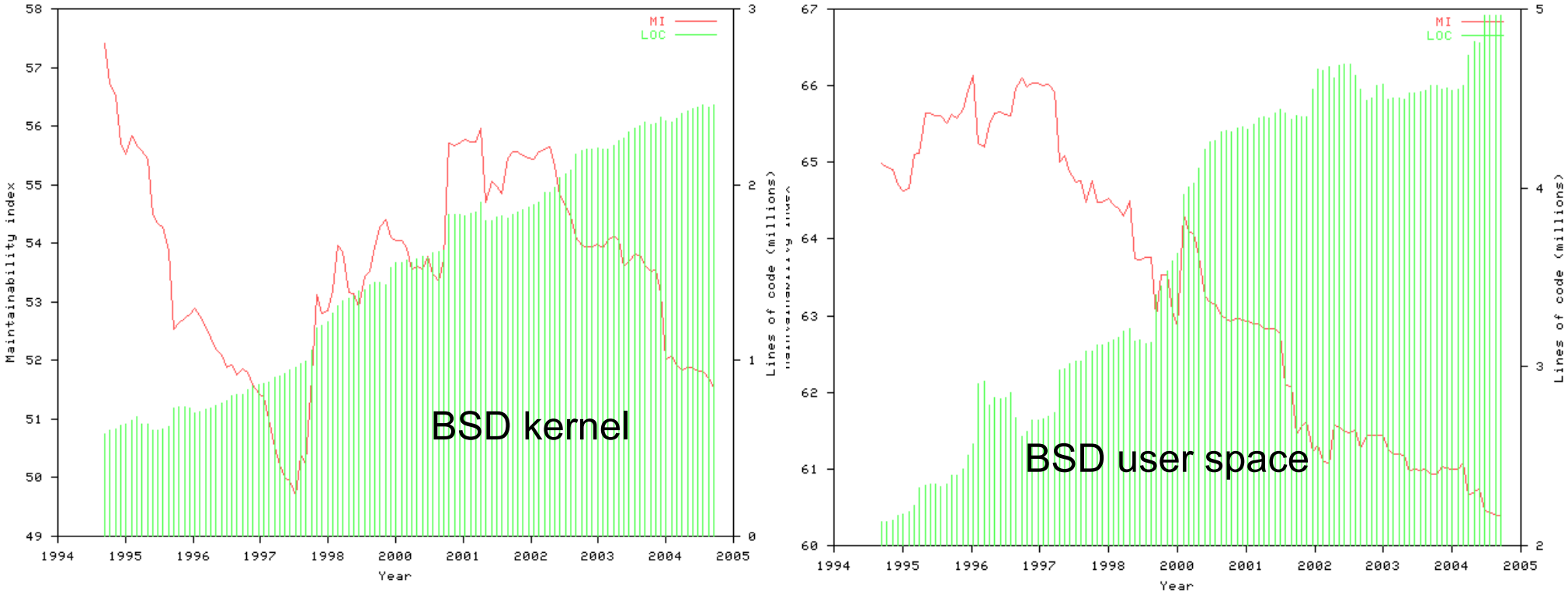
- La *complessità ciclomatica estesa* tiene conto della complessità aggiuntiva derivante dalle espressioni condizionali complesse
- Un'espressione condizionale complessa contiene condizioni multiple separate da OR o AND.
- Quando un costrutto di controllo contiene un'espressione logica con OR e/o AND, la misura di Complessità Ciclomatica Estesa aumenta di uno per ciascun operatore logico impiegato nel costrutto
- Vedere gli esempi
  http://www.crosstalkonline.org/storage/issue-archives/2001/200108/200108-Welker.pdf

Strumenti

- Esistono parecchi strumenti in grado di supportare la manutenzione del sw
- Molti registrano misure utili per valutare l'indice di manutenibilità
- Esempio: plugin *metrics* di Eclipse
  http://metrics.sourceforge.net


Java+ - MoneyBag.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run CodePro Window Help
Package Explorer+
Outline
miscellaneous
money
Customer.java
CustomerFactory.jav
CustomerUsage.java
IMoney.java
Money.java
MoneyBag.java
Test.java
z ++;
++ z;
StringBuffer buff = new StringBuffer();
HashMap map = new HashMap();
if (list instanceof List[]) return null;
if (returnNull) {
return null;
} else {
String s = null;
Code Coverage
Code Coverage -- com.instantiations.example.TestAll
CodePro JUnit Demo (87.1% , 26)
com.instantiations.example.miscellaneou
com.instantiations.example.money (80.8
Customer.java (100.0% , 0)
Money.java (91.0% , 4)
MoneyBag.java (72.9% , 19)
Simple.java (95.0% , 1)
com.instantiations.example.pattern (95.
com.instantiations.example.stack (100.0
Metrics
TestCase Outline
Metrics (CodePro Demo at 1/19/05 9:40 PM)
Name
Value
Average Lines Of Code Per M...
5.91
Average Number of Construc...
0.50
Average Number of Fields Pe...
1.00
Average Number of Methods ...
3.36
com.instantiations
8.00
com.instantiations.guitest
12.00
com.instantiations.jrf.demo
2.70
Average Number of Parameters
0.44
Types by Method Count
Types Grouped by Method Count
Method Count
MoneyBag.java
void appendBag(MoneyBag aBag) {
for (Enumeration e= aBag.fMonies.
appendMoney((Money)e.nextElem
void appendMoney(Money aMoney) {
if (aMoney.isZero()) return;
IMoney old= findMoney(aMoney.curr
if (old == null) {
Details
History
Classes: 4 / 4 100.0%
Methods: 39 / 40 97.5%
Lines: 97 / 121 80.2%
Blocks: 84 / 106 79.2%
Instructions: 399 / 494 80.8%
Audit
Audit (CodePro Demo at 1/19/05
No Abstract Methods [2]
Numeric Literals [27]
Obsolete Modifier Usage
Parenthesize Condition i
Return Value [3]
Return zero-length a
Return zero-length a
Return zero-length array instead of null (Test.java - Line 45)
Static Field Naming Convention [3]
Go to File
Copy to Clipboard
Export Violations
Explain...
Fix Violation
Disable Return Value Rule
Configure Rule...
Create Task
Remove Violations
Delete
Replace with empty array
Disable rule for line
Disable all rules for line
Disable rule for file
Disable all rules for file
Description
nstead of null
y instead of null so that
ude additional null checks
and to avoid the possibility of null pointer
exceptions.

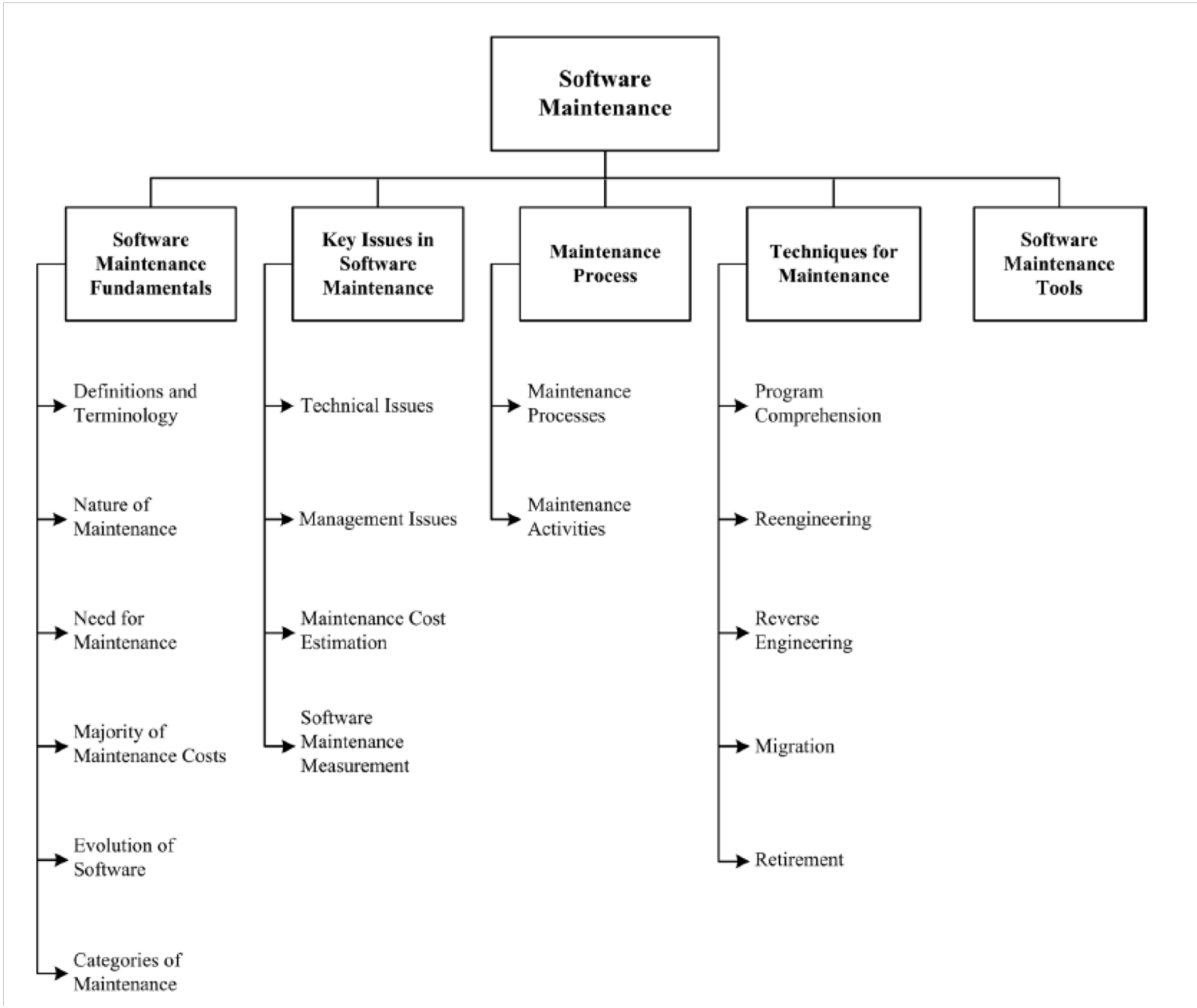
Ricerche sulla manutenibilità



- Analisi 2005 di MI vs LOC su FreeBSD Unix, kernel e user space. D. Spinellis 2005

www.spinellis.gr/blog/20050204/index.html

La manutenzione in SWEBOK



Lettura raccomandata

McCabe & Thomas, A Complexity Measure,
IEEE Trans. on Sw Engineering,1976

Pubblicazioni di ricerca sulla manutenzione del software

- Int. Conf. on Software Maintenance
- Working Conf. on Reverse Engineering (WCRE)
- European Conf. on Software Maintenance and Reengineering (CSMR)
- Journal of Software Maintenance and Evolution: Research and Practice

Domande di autotest

- Cos’è il numero ciclomatico?
- Cos’è la software science?
- Quali indicatori di processo sono importanti per valutare la manutenibilità del software?

Riferimenti

Capitolo 6 del SWEBOK: “Software maintenance”, 2004

IEEE Standard 1219-1998 for Software Maintenance

Mens, *Software Evolution*, Springer, 2008

Lehman & Belady, *Program evolution : processes of software change*, AP 1985

Pigoski, *Practical Software Maintenance*. Wiley 1996

Lehman e altri, Metrics and Laws of Software Evolution - The Nineties View, *Int. Conf. on Sw Metrics*, 1997

Oman e Hagemeister, Metrics for assessing a software system's maintainability, *Int. Conf. on Software Maintenance*, 1992

Welcher, The Software Maintainability Index Revisited, *Crosstalk* 2001

Siti

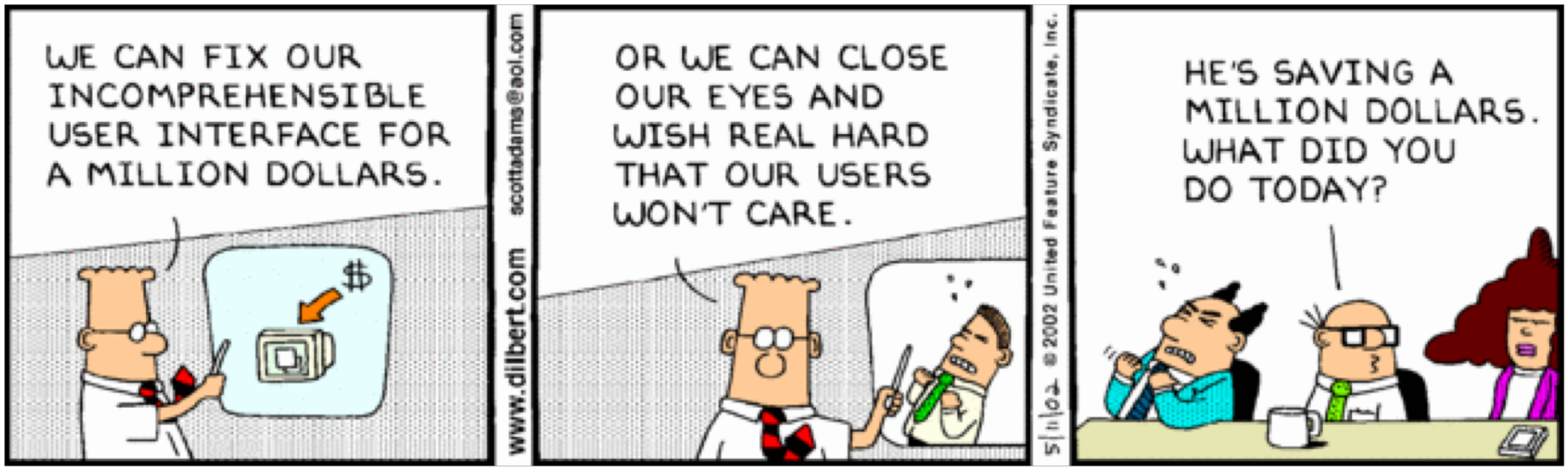
`www.stateofflow.com/projects/16/eclipsemetrics`

`pmd.sourceforge.net` strumento open source per analisi codice Java

`radon.readthedocs.org/en/latest/index.html` altro strumento

`www.moosetechnology.org` piattaforma per analisi del sw

Domande?


WE CAN FIX OUR INCOMPREHENSIBLE USER INTERFACE FOR A MILLION DOLLARS.
$
www.dilbert.com scottadams@aol.com
OR WE CAN CLOSE OUR EYES AND WISH REAL HARD THAT OUR USERS WON'T CARE.
© 2002 United Feature Syndicate, Inc.
HE'S SAVING A MILLION DOLLARS. WHAT DID YOU DO TODAY?