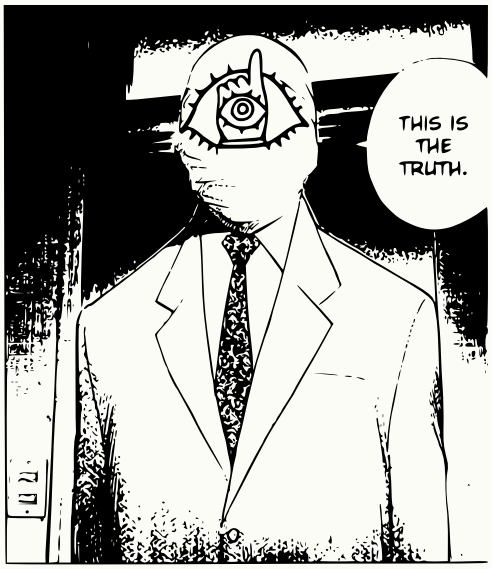
Marco Casu

Ingegneria del Software

Sapienza
Università di Roma

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Questo documento è distribuito sotto la licenza GNU, è un resoconto degli appunti (eventualmente integrati con libri di testo) tratti dalle lezioni del corso di Ingegneria del Software per la laurea triennale in Informatica. Se dovessi notare errori, ti prego di segnalarmeli.



INDICE

CAPITOLO

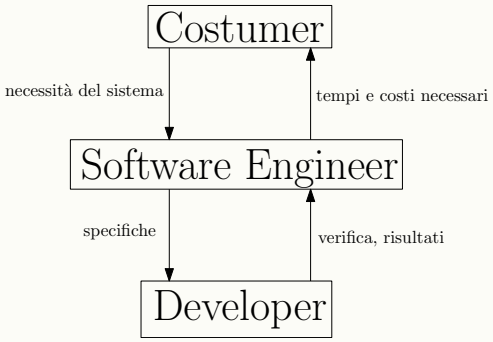
1 INTRODUZIONE

Quando si vuole produrre software complesso, non terminante e di grandi dimensioni, è necessario ingegnerizzare il metodo di sviluppo ed astrarre il modello del software. L'obiettivo dell'ingegneria del software non riguarda il codice stesso, ma la definizione dell'architettura e delle specifiche del sistema. Vanno definiti i processi di sviluppo.

È cruciale l'analisi dei requisiti, e la modellizzazione delle specifiche di sistema, UML è un linguaggio che permette di descrivere la *dinamica* ed il comportamento del modello. Una volta definiti i requisiti, si affronta la fase di *planning*.

Un modello computazionale che risulterà utile prende il nome di *digital twin*, ossia una copia del software sulla quale è possibile testare i vari *scenari operativi*, tramite la generazione di test automatici. Essendo il sistema discreto, a stati non necessariamente finiti, è possibile descriverlo tramite una catena di Markov.

Nella produzione software prendono parte 3 principali attori



1.1 Modellazione

Il linguaggio UML è utilizzato per modellare sia i requisiti che il sistema stesso, esso è composto da differenti diagrammi

- activity
- use case
- sequence

- class
- state
- context

Si consideri il seguente esempio di un sistema di gestione di una clinica psichiatrica, lo schema in figura 1.1, è il context diagram, e rappresenta l'insieme dei serivizi che il sistema offre. Un activity diagram descrive

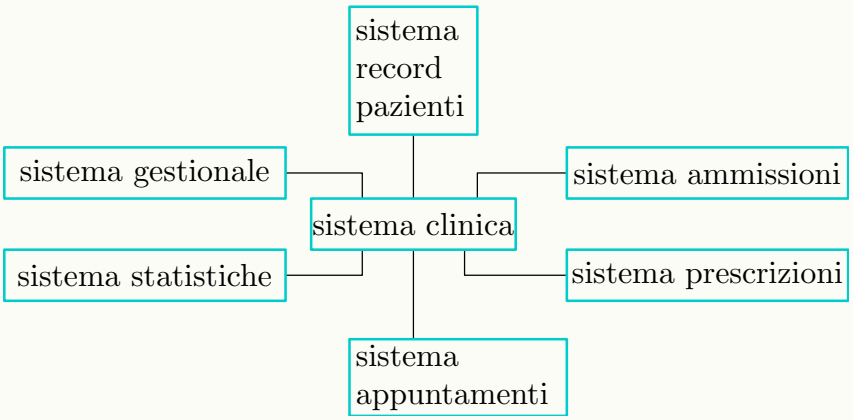


Figura 1.1: context diagram

l'evoluzione di una certa attività/task che il sistema deve poter implementare, si considere l'esempio in figura 1.2 riguardante l'inserimento di un paziente nella clinica.

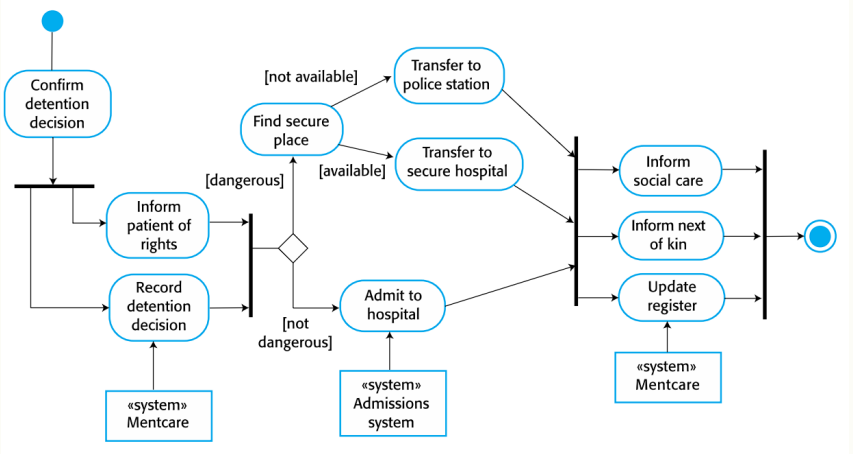


Figura 1.2: activity diagram

1.2 Catene di Markov per la Progettazione

La moderna progettazione di sistemi complessi si basa sul concetto di digital twin, in modo da poter definire un modello che simula il sistema su cui verranno eseguiti determinati esperimenti al fine di collaudo prima della progettazione vera e propria. Questo capitolo si concetra sull'aspetto *predittivo* dei digital twin : Si vuole utilizzare il modello per fare predizioni sull'andamento della progettazione.

Definizione (DTMC) : Una **Catena di Markov Discreta** è una tupla (U, X, Y, p, g) tale che

- U è un insieme (possibilmente vuoto, finito o non) denotato *input/ingresso* del sistema.
- X è un insieme non vuoto (finito o non) denotato *stato* del sistema.

- Y è un insieme non vuoto (finito o non) denotato *output/uscita* del sistema.
- p è la *probabilità di transizione*, è una funzione

  $$p : X \times X \times U \to [0, 1]$$

  tale che $p(x'|x, u)$ rappresenta la probabilità che la DTMC si sposti dallo stato x allo stato x' quando l'ingresso è u. Inoltre, per ogni $\tilde{x} \in X$ e per ogni $\tilde{u} \in U$, si ha che

  $$\int_{x' \in X} p(x'|\tilde{x}, \tilde{u}) = 1$$

  informalmente, per ogni possibile coppia stato-ingresso, esiste uno stato successivo.
- g è una funzione

  $$g : X \to Y$$

  detta *funzione di output*.

Data una catena di markov M, denotiamo $M(U), M(X), M(Y)$ rispettivamente : gli ingressi, gli stati e le uscite.

Esempio tempi di completamento : Si consideri il processo di sviluppo mostrato in figura 1.3, si vuole calcolare la probabilità che il processo termini in esattamente 10 mesi.

Si identificano i possibili cammini la cui somma dei sotto processi impiega 10 mesi

$$0 \to 1 \to 1 \to 2 \to 3 \to 4$$
$$0 \to 1 \to 2 \to 3 \to 3 \to 4$$

La probabilità che ciò accada è la somma delle probabilità sui due percorsi

$$\begin{matrix} 1 \cdot 0.3 \cdot 0.7 \cdot 0.6 \cdot 0.6 = 0.0756 \\ 1 \cdot 0.7 \cdot 0.6 \cdot 0.1 \cdot 0.6 = 0.02520 \end{matrix} \implies p(\text{finire in 10 mesi}) = 0.1008$$

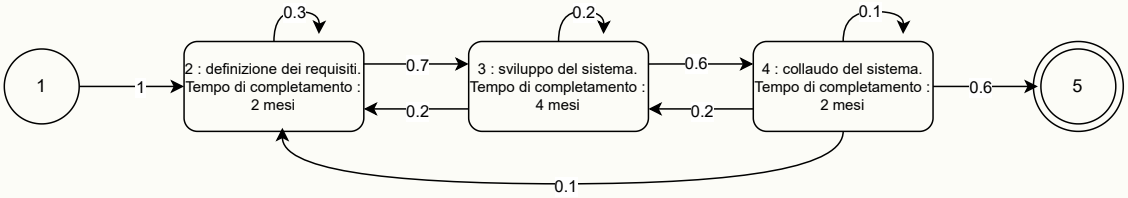


Figura 1.3: Esempio 1

Qual'è invece la probabilità di finire il processo in esattamente 12 mesi? Si identificano i cammini

$$0 \to 1 \to 1 \to 1 \to 2 \to 3 \to 4$$
$$0 \to 1 \to 1 \to 2 \to 3 \to 3 \to 4$$
$$0 \to 1 \to 2 \to 2 \to 3 \to 4$$
$$0 \to 1 \to 2 \to 3 \to 3 \to 3 \to 4$$

la probabilità è

$$1 \cdot 0.3 \cdot 0.3 \cdot 0.7 \cdot 0.6 \cdot 0.6 +$$
$$1 \cdot 0.3 \cdot 0.7 \cdot 0.6 \cdot 0.1 \cdot 0.6 +$$
$$1 \cdot 0.7 \cdot 0.2 \cdot 0.6 \cdot 0.6 +$$
$$1 \cdot 0.7 \cdot 0.6 \cdot 0.1 \cdot 0.1 \cdot 0.6 =$$
$$0.022680 + 0.007560 \cdot 0.05040 \cdot 0.002520 = 0.08316$$

Esempio costi di completamento : Si consideri il processo di sviluppo mostrato in figura 1.4, si vuole calcolare la probabilità che il processo abbia un costo toale di esattamente 80.000 euro.

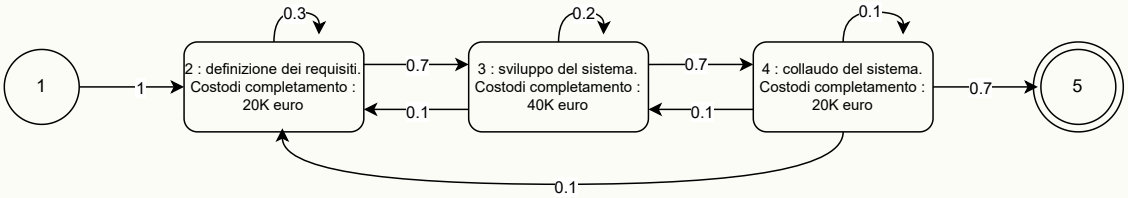


Figura 1.4: Esempio 2

L'unico cammino possibile è

$$0 \to 1 \to 2 \to 3 \to 4$$

La probabilità è

$$1 \cdot 0.7 \cdot 0.7 \cdot 0.7 = 0.343$$

Esempio costi di design : Si consideri il processo di sviluppo mostrato in figura 1.5, quale deve essere il minimo valore di p tale che la probabilità che il processo termini in esattamente 8 mesi sia almeno 0.5?

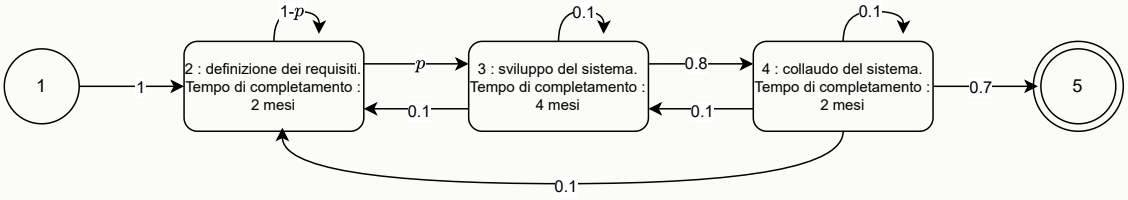


Figura 1.5: Esempio 3

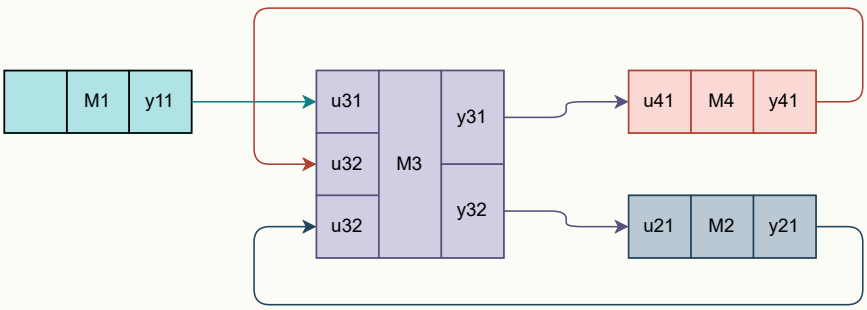
L'unico cammino possibile è

$$0 \to 1 \to 2 \to 3 \to 4$$

la cui probabilità deve essere

$$1 \cdot p \cdot 0.8 \cdot 0.7 \geq 0.5 \implies$$
$$p \geq \frac{0.5}{0.8 \cdot 0.7} = 0.8928$$

Una **rete di catene di Markov** è un insieme di catene di Markov interconnesse tramite gli ingressi e le uscite.



Inoltre, una rete di catene di Markov è rappresentabile come una catena di Markov singola, si considerino le catene di Markov K, P in figura 1.6, formalmente

$$K = (U_1 \times V, X_1, Y_1, p_1, g_1)$$
$$P = (U_2, X_2, W \times Z, p_2, (g_{2_w}, g_{2_z}))$$

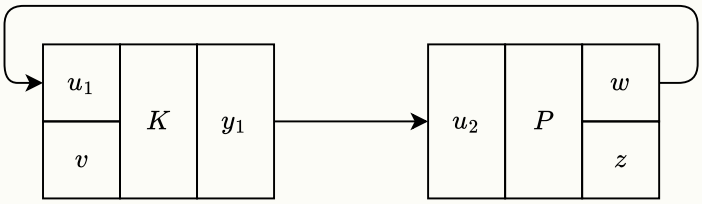


Figura 1.6: K e P interconnesse

Nota : u_1 rappresenta un elemento dell'insieme U_1 (ingresso di K), $v \in V$, $y_1 \in Y_1 \dots$ analogo per gli altri elementi.

Un ingresso di K è un uscita di P

$$M_1(U_1) = M_2(W)$$

e l'ingresso di P è l'uscita di K

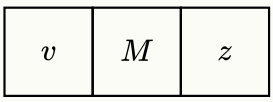
$$M_2(U_2) = M_1(Y_1)$$

Si può definire il sistema globale equivalente

$$M = (V, X_1 \times X_2, Z, p, g)$$

Dove

- $p((x_1', x_2')|(x_1, x_2), v) = p_1(x_1'|x_1, (g_{2_w}(x_2), v)) \cdot p_2(x_2'|x_2, g_1(x_1))$
- $g(x_1, x_2) = g_{2_z}(x_2)$



1.3 Implementazione delle DTMC

In questa sezione verrà trattata una possibile implementazione delle DTMC in *C++*. Si assume che il lettore sia a conoscenza delle regole di scrittura e strutturazione degli header e file C.

Una DTMC sarà implementata in due file
`dtmc.h`
`dtmc.cpp`
Nel file `dtmc.h` come prima cosa, si definisce il *tipo* delle variabili di ingresso, stato, e uscita, tramite il costrutto `typedef` .

```
1 typedef double state_type;
2 typedef double input_type;
3 typedef double output_type;
```

Va poi definita la classe `DTMC` le cui istanze rappresenteranno proprio le catene di Markov, gli elementi privati di tale classe saranno 2

- Un puntatore ad una classe `RandGen` definita in un'apposito file `randgen.cpp`
- Una funzione `dtmc_body` che verrà definita nel file `dtmc.cpp`

```
class DTMC {

private:
    RandGen *p;  // pointer to random generator class
    void dtmc_body(RandGen *p);
```

Fra gli elementi pubblici della classe ci saranno poi le dimensioni dei vettori di ingresso, stato ed uscita [di default, assumeremo che tali vettori siano in $\mathbb{R}^2$], insieme a delle variabili che rappresentano lo stato corrente, l'ingresso e l'uscita. Ci saranno poi due costruttori, uno più esplicito dell'altro

- Il primo costruttore riceve in ingresso esclusivamente il puntatore all'oggetto di tipo `RandGen`, lasciando invariate le dimensioni dei vettori di stato, ingresso ed uscita a 2.
- Il secondo costruttore è più esplicito e permette la modifica delle dimensioni dei vettori di stato, ingresso ed uscita.

```
public:
    int STATE_SIZE = 2; // state size
    int INPUT_SIZE = 2; // input size
    int OUTPUT_SIZE = 2; // output size

    state_type *x;  // present state
    input_type *u;  // input
    output_type *y;  // output

    DTMC(RandGen *p);
    DTMC(int xsize, int usize, int ysize, RandGen *p);
```

Ci sono poi le varie funzioni della classe, precisamente per

- inizializzare la catena con lo stato iniziale
- calcolare il prossimo stato dato lo stato attuale e l'ingresso
- dato uno stato restituisce l'uscita del sistema

Ci sono poi due funzioni di servizio per la visualizzazione dello stato in console (oppure su un file).

```
    // define initial state
    void init();

    // update state
    void next();

    // update output
    void output();

    // print log on stdout
    void printlog(int t);

    // print log on file
    void fprintflog(FILE *fp, int t);
```

Verrà considerata adesso la definizione delle funzioni dichiarate, precisamente, nel file `dtmc.cpp`. Ovviamente vanno incluse le dichiarazioni.

```
#include "dtmc.h"
using namespace std;
```

La funzione `dtmc_body` verrà richiamata dai costruttori per definire ed inizializzare la catena di Markov. Deve ricevere in input il puntatore al generatore di numeri casuali ed assegnarlo alla variabile interna (privata). Dopo aver fatto ciò viene fatto un controllo sulle dimensioni del vettore di stato che devono essere strettamente maggiori di zero.

```
void dtmc_body(RandGen *myp) {
    p = myp;
    if (STATE_SIZE <= 0)
    {
     fprintf(stderr, "DTMC(): Error: state size cannot be 0\n");
     exit(1);
    }
```

A questo punto, bisogna inizializzare il vettore di stato, dato che le dimensioni non sono note a priori, sarà necessario allocare dinamicamente lo spazio.

```
    x = (state_type *) malloc(sizeof(state_type)*STATE_SIZE);

    if (x == NULL)
        {
        fprintf(stderr, "DTMC(): memory allocation for state failed\n");
        exit(1);
    }
```

Si prosegue analogamente con l'ingresso e l'uscita, si ricordi che una DTMC può avere ingresso nullo, ma deve necessariamente esserci un uscita.

```
    if (INPUT_SIZE <= 0)
    {
        u = NULL;
    }
    else
    {
        u = (input_type *) malloc(sizeof(input_type)*INPUT_SIZE);
        if (u == NULL)
        {
            fprintf(stderr, "DTMC(): memory allocation
                                for input failed\n");
            exit(1);
        }
    }

    if (OUTPUT_SIZE <= 0)
    {
        fprintf(stderr, "DTMC(): Error: output size cannot be 0\n");
        exit(1);
    }

    y = (output_type *) malloc(sizeof(output_type)*OUTPUT_SIZE);
    if (y == NULL)
    {
        fprintf(stderr, "DTMC(): memory allocation
                            for output failed\n");
        exit(1);
    }
}
```

A tal punto i due costruttori possono richiamare tale funzione per inizializzare la catena.

```
DTMC(RandGen *myp)  {
    dtmc_body(myp);
}

DTMC(int xsize, int usize, int ysize, RandGen *myp)  {
    STATE_SIZE = xsize;
    INPUT_SIZE = usize;
    OUTPUT_SIZE = ysize;
```

```
    dtmc_body(myp);
}
```

La funzione di inizializzazione inizializza a zero il vettore di stato.

```
void init() {
    /* initial state */
    for (int i = 0; i < STATE_SIZE; i++)
    {
        x[i] = 0;
    }
}
```

La funzione `next` dovrà calcolare lo stato successivo

```
void next() {
    /* initial state */
    for (int i = 0; i < STATE_SIZE; i++)
    {
        x[i] = p -> get_random_double();
    }
}
```

La funzione `get_random_double` è definita in `RandGen` e restituisce un double casuale compreso fra 0 e 1. In questo caso specifico, lo stato è un vettore di numeri reali, quindi è una DTMC a stati infiniti. Si definisce poi la funzione in output, in questo specifico caso, l'output sarà identico allo stato.

```
void output() {
    /* initial state */
    for (int i = 0; i < OUTPUT_SIZE; i++)
    {
        y[i] = x[i];
    }
}
```

Ovviamente le dimensioni devono combaciare. Si definiscono poi le funzioni di servizio che si occuperanno di eseguire un print dello stato della catena di Markov in formato CSV, così poi da rendere facile la generazione di un grafico.

```
void printlog(int t) {
    /* t e' il tempo corrente */
    int i;
    printf("%d ", t);
    for (i = 0; i < STATE_SIZE; i++)
    {
        printf("%lf ", x[i]);
    }
    for (i = 0; i < OUTPUT_SIZE; i++)
    {
        printf("%lf ", y[i]);
    }
    for (i = 0; i < INPUT_SIZE; i++)
    {
        printf("%lf ", u[i]);
    }
    printf("\n");
}
```

La funzione `fprintflog` è analoga ma scrive su un file.

Si da ora una descrizione della classe `RandGen`, definita nei file
`randgen.h`
`randgen.cpp`

Il file header è molto semplice, contiene la dichiarazione del costruttore, una variabile (seed) e due funzioni per la generazione di numeri double e interi.

```
class RandGen{
public:
    RandGen();
    unsigned int seed;
    double get_random_double();
    int get_random_int();
}
```

Il seed è un intero che condiziona la generazione dei valori. Il file `randgen.cpp` contiene ovviamente le definizioni ed utilizza una serie di funzioni già preesistenti del C++ definite nella libreria matematica. Oltre ad includere il file header, si utilizza un oggetto già definito per la generazione del seed. Dopo di ciò, si inizializzano arbitrariamente delle distribuzioni. Nell'esempio seguente, per i double i valori che possono essere estratti sono compresi fra 0 ed 1, per gli interi da -10 a 10.

```
#include "randgen.h"
random_device myRandomDevice;
unsigned int myseed = myRandomDevice();

default_random_engine myRandomEngine(myseed);
uniform_int_distribution<int> myUnifIntDist(-10,10);
uniform_real_distribution<double> myUnifRealDist(0.0,1.0);
RandGen(){
    seed=myseed;
}
```

Si definiscono poi le funzioni che generano il numero casuale.

```
double get_random_double() {
  return(myUnifRealDist(myRandomEngine));
}

int RandGen::get_random_int() {
  return(myUnifIntDist(myRandomEngine));
}
```

A questo punto, si è definita in tutto e per tutto una catena di markov a tempo discreto, si vuole scrivere un *simulatore* che si occupi di istanziare la classe e di eseguire dei test. Si definiscono quindi due file
`simulator.h`
`simulator.cpp`
Come prima cosa nel file header si definisce la classe `Simulator` i cui oggetti faranno "muovere" la/le catena/e di Markov. È una classe con una sola funzione pubblica.

```
#include "main.h"
class Simulator {
public:
  // run simulation with horizon T
  void run(int T);
};
```

La funzione `run` prende come argomento "l'orizzonte", ossia il numero di step finiti che la catena di Markov deve compiere.

Il simulatore è in tutto e per tutto la funzione `run` definita nel file `simulator.cpp`, che definisce come prima cosa gli elementi necessari alla DTMC.

```
void Simulator(int T){
    RandGen p;
    DTMC mc1(&p);
    int t=0;
    FILE *fp;
```

```
    printf("seed : %u\n",p.seed); //servizio
    printf("inizialization : t=  %d\n",t); //servizio
    mc1.init();
    fp=fopen("logfile.csv","w");
```

Il tempo `t` si inizializza a zero. Il file `fp` sarà quello in cui verrà scritto il log della catena. A tal punto va eseguita la simulazione vera e propria eseguendo precisamente `T` step sulla catena di Markov, scrivendo lo stato sul log ad ogni passo.

```
    for (t = 0; t <= T; t++)
    {
        mc1.output();
        mc1.u[0] = t;  mc1.u[1] = mc1.y[1] + 1;
        mc1.printlog(t);
        mc1.fprintflog(fp, t);
        mc1.next();
        #if 0
            printf("get state of system at time %d\n", t);
            printf("get output of system at time %d\n", t);
            printf("update inputs with outputs at time %d\n", t);
            printf("update state of system with
                    inputs at time %d\n\n", t);
        #endif
    }
    fclose(fp);
    }
```

Ad ogni passo, il vettore in input assume il valore (tempo,uscita + 1). L'output si ricordi essere funzione esclusivamente dello stato e non dell'input, è quindi perfettamente legittimo utilizzare come input l'uscita del sistema, in tal modo detto "a retroazione".

$$u(t) = \begin{bmatrix} t \\ y(t-1)+1 \end{bmatrix}$$

Il progetto può essere poi compilato con un makefile, un possibile makefile generico è il seguente.

```
CC=g++
#CFLAGS=-std=c++11 -lm -I. -I/usr/include/postgresql -lpq
#CFLAGS=-std=c++11 -I. -I/usr/include/postgresql -lpq -lm
CFLAGS=-std=c++11 -I.  -lm
DEPS = $(wildcard *.h)
objects := $(patsubst %.cpp,%.o,$(wildcard *.cpp))
%.o:    %.cpp $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

main:   $(objects)
    $(CC) -o main $(objects) $(CFLAGS)
clear:
    rm *.o
clearall:
    rm main ; rm *.o
```

1.4 Raccolta dei File

1.4.1 `main.h`

```
#ifndef main_h
#define main_h

using namespace std;

#include <iostream>
#include <random>

#include "randgen.h"
#include "dtmc.h"
#include "simulator.h"

#if 0
#include "clock.h"
#include "logger.h"
#include "gtime.h"
#include "timer.h"
#include "basic.h"
#include "system.h"

#include "ctr.h"
#include "plant.h"
#include "root.h"
#endif

#define DEBUG 1000

#define HORIZON 10

#if 0
extern Logger val2log;
extern GlobalTime gtime;
extern Clock ck;
extern random_device myRandomDevice;
extern unsigned int seed;
extern default_random_engine myRandomEngine;
extern uniform_int_distribution<int> myUnifIntDist;
extern uniform_real_distribution<double> myUnifRealDist;
#endif

#if 0
    random_device myRandomDevice;
    unsigned int seed = myRandomDevice();
// Initialize a default_random_engine with the seed
    default_random_engine myRandomEngine(seed);
// Initialize a uniform_int_distribution to produce values between -10
   and 10
    uniform_int_distribution<int> myUnifIntDist(-10, 10);
// Initialize a uniform_real_distribution to produce values between 0
   and 1
    uniform_real_distribution<double> myUnifRealDist(0.0, 1.0);
#endif
#endif
```

1.4.2 dtmc.h

```
#ifndef dtmc_h
#define dtmc_h

#include "main.h" /*include il file dove e'
                    definita la classe RandGen*/

typedef double state_type;
typedef double input_type;
typedef double output_type;

class DTMC {

private:

    RandGen *p;  // pointer to random generator class

    void dtmc_body(RandGen *p);

public:

    int STATE_SIZE = 2; // state size
    int INPUT_SIZE = 2; // input size
    int OUTPUT_SIZE = 2; // output size

    state_type *x;  // present state
    input_type *u;  // input
    output_type *y;  // output

    DTMC(RandGen *p);
    DTMC(int xsize, int usize, int ysize, RandGen *p);

    // define initial state
    void init();

    // update state
    void next();

    // update output
    void output();

    // print log on stdout
    void printlog(int t);

    // print log on file
    void fprintflog(FILE *fp, int t);
};

#endif
```

1.4.3 dtmc.cpp

```
#include "main.h"
#include "dtmc.h"

void dtmc_body(RandGen *myp) {
    p = myp;
    if (STATE_SIZE <= 0)
    {
        fprintf(stderr, "DTMC(): Error: state size cannot be 0\n");
        exit(1);
    }
    x = (state_type *) malloc(sizeof(state_type)*STATE_SIZE);
    if (x == NULL)
    {
        fprintf(stderr, "DTMC(): memory allocation for state failed\n");
        exit(1);
    }
    if (INPUT_SIZE <= 0)
    {
        u = NULL;
    }
    else
    {
        u = (input_type *) malloc(sizeof(input_type)*INPUT_SIZE);
        if (u == NULL)
        {
            fprintf(stderr, "DTMC(): memory allocation
                             for input failed\n");
            exit(1);
        }
    }
    if (OUTPUT_SIZE <= 0)
    {
        fprintf(stderr, "DTMC(): Error: output size cannot be 0\n");
        exit(1);
    }
    y = (output_type *) malloc(sizeof(output_type)*OUTPUT_SIZE);
    if (y == NULL)
    {
        fprintf(stderr, "DTMC(): memory allocation
                         for output failed\n");
        exit(1);
    }
}


DTMC(RandGen *myp)  {
    dtmc_body(myp);
}

DTMC(int xsize, int usize, int ysize, RandGen *myp)  {
    STATE_SIZE = xsize;
    INPUT_SIZE = usize;
    OUTPUT_SIZE = ysize;
    dtmc_body(myp);
}

```

```
57 void init() {
58     int i;
59     for (i = 0; i < STATE_SIZE; i++)
60         x[i] = 0;
61 }
62 void next() {
63     int i;
64     for (i = 0; i < STATE_SIZE; i++)
65         x[i] = p -> get_random_double();
66 }
67 void output() {
68     int i;
69     for (i = 0; i < OUTPUT_SIZE; i++)
70         y[i] = x[i];
71 }
72
73 void printlog(int t) {
74     int i;
75     printf("%d ", t);
76     for (i = 0; i < STATE_SIZE; i++)
77     {
78         printf("%lf ", x[i]);
79     }
80     for (i = 0; i < OUTPUT_SIZE; i++)
81     {
82         printf("%lf ", y[i]);
83     }
84     for (i = 0; i < INPUT_SIZE; i++)
85     {
86         printf("%lf ", u[i]);
87     }
88     printf("\n");
89 }
90
91 void fprintflog(FILE *fp, int t) {
92     FILE *fp;
93     int i;
94     fp = fopen(filename, "w");
95     fprintf(fp, "%d ", t);
96     for (i = 0; i < STATE_SIZE; i++)
97     {
98         fprintf(fp, "%lf ", x[i]);
99     }
100    for (i = 0; i < OUTPUT_SIZE; i++)
101    {
102        fprintf(fp, "%lf ", y[i]);
103    }
104    for (i = 0; i < INPUT_SIZE; i++)
105    {
106        fprintf(fp, "%lf ", u[i]);
107    }
108    fprintf(fp, "\n");
109 }
```

1.4.4 randgen.h

```
#ifndef randgen_h
#define randgen_h
#include "main.h"

class RandGen {
public:
    RandGen();
    unsigned int seed;
    double get_random_double();
    int get_random_int();
};
#endif
```

1.4.5 randgen.cpp

```
#include "main.h"
#include "randgen.h"

random_device myRandomDevice;
unsigned int myseed = myRandomDevice();

default_random_engine myRandomEngine(myseed);
uniform_int_distribution<int> myUnifIntDist(-10, 10);
uniform_real_distribution<double> myUnifRealDist(0.0, 1.0);
RandGen() { seed = myseed; }

double get_random_double() {
    return(myUnifRealDist(myRandomEngine));
}

int get_random_int() {
    return(myUnifIntDist(myRandomEngine));
}
```

CAPITOLO

2 PLANNING DEL PROGETTO

Il planning di un progetto, consiste nel suddividere il carico di lavoro in diverse parti, da assegnare ai vari membri del team, cercando di prevedere possibili problemi che potrebbero insorgere durante lo sviluppo, pensando e preparando eventuali modi per risolverli.

Il *piano del progetto* viene preparato all'inizio dei lavori, viene utilizzato per comunicare ai vari membri del team ed al cliente come esso è stato suddiviso. Le fasi in cui il planning viene definito sono

- Durante la proposta, quando avviene la contrattazione con il cliente riguardo lo sviluppo del software
- Durante la fase di avvio dei lavori, quando si decide come e a chi assegnare il lavoro, e quali risorse dovranno essere allocate
- Periodicamente durante lo sviluppo, monitorando ed appositamente modificando i piani in base all'andamento del progetto e alle esperienze pregresse

Lo scopo del planning è quello di avere un idea chiara sul progetto in modo che si possa decidere un prezzo d'accordo con il cliente, valutando e stimando quanto denaro sarà necessario per lo sviluppo considerando variabili del tipo

- costo dei dipendenti
- costo dell'hardware necessario
- costo del software

Durante la fase di planning, sono noti i requisiti del sistema, ma non è ancora chiara la struttura del software, tanto meno la sua implementazione, il planning deve essere preciso a sufficienza per far si che sia possibile definire il budget e lo staff necessario. Inoltre bisogna definire i meccanismi con la quale verrà monitorato lo sviluppo.

2.1 Sviluppo Plan-Driven

Con *plan driven development*, si intende un approccio all'ingegneria del software in cui i processi di sviluppo sono programmati in principio, in maniera dettagliata. Si basa sulle tecniche di gestione, tipiche dei progetti ingegneristici, e sulle tecniche "classiche" di gestione di grandi progetti software.

Un project plan ha l'obiettivo di definire e monitorare il lavoro da svolgere, in che modo deve essere

svolto, da chi, e quali prodotti sono necessari. È scopo del manager, servirsi di un project plan (che da ora chiameremo semplicemente "piano") per supportare le decisioni da prendere durante lo sviluppo, e per misurarne il progresso.

- Un lato favorevole di tale approccio, è che una pianificazione a monte permette di aggirare problemi organizzativi in principio, determinando potenziali problemi e dipendenze da soddisfare prima che il progetto sia avviato, piuttosto che durante la lavorazione.

  meglio prevenire che curare

- Un lato sfavorevole, è che molte decisioni prese in principio vanno riviste dati possibili cambiamenti dell'ambiente in cui il software deve essere adoperato.

Un piano deve definire, le risorse disponibili per il progetto, la suddivisione del carico di lavoro, una schedule per portare a termine il lavoro. Precisamente, un piano consiste nelle seguenti sezioni

1. introduzione
2. organizzazione del progetto
3. analisi dei rischi
4. risorse hardware e software necessarie
5. struttura di scomposizione del lavoro
6. schedule del progetto
7. meccanismi di monitoraggio e report

Ci sono inoltre altri tipi di "piano" che possono essere aggiunti a quello principale come supporto:

Piano	Descrizione
Configuration managment plan	descrive la configurazione delle procedure di gestione, la loro struttura ed il loro utilizzo
Deployment plan	descrive come il software deve essere integrato con l'apposito hardware di riferimento del cliente, con eventuali piani di migrazione dei dati in uso su sistemi precedentemente adoperati
Maintenance plan	predizione dei requisiti, costi e lavoro per la manutenzione del software
Quality plan	descrizione delle procedure e standard di qualità utilizzati nel progetto
Validation plan	descrizione degli approcci, risorse e schedule utilizzati per la convalida del sistema

Il planning del progetto è un processo *iterativo* che viene sottoposto ad inevitabili cambiamenti durante lo sviluppo progressivo. Con l'aumentare delle informazioni relative al sistema durante lo sviluppo, è doveroso revisionare i requisiti iniziali, dei cambiamenti nel business possono portare a grandi cambiamenti nei requisiti del progetto, portando ad un eventuale ri-pianificazione totale.

Assuzioni del planning

- Le assunzioni da fare durante la definizione del project plan non devono essere ottimistiche, bensì realistiche.
- Durante lo sviluppo insorgeranno inevitabilmente dei problemi che causeranno dei ritardi nella consegna.
- Le assunzioni iniziali e lo scheduling saranno inevitabilmente soggetti a problemi inaspettati.

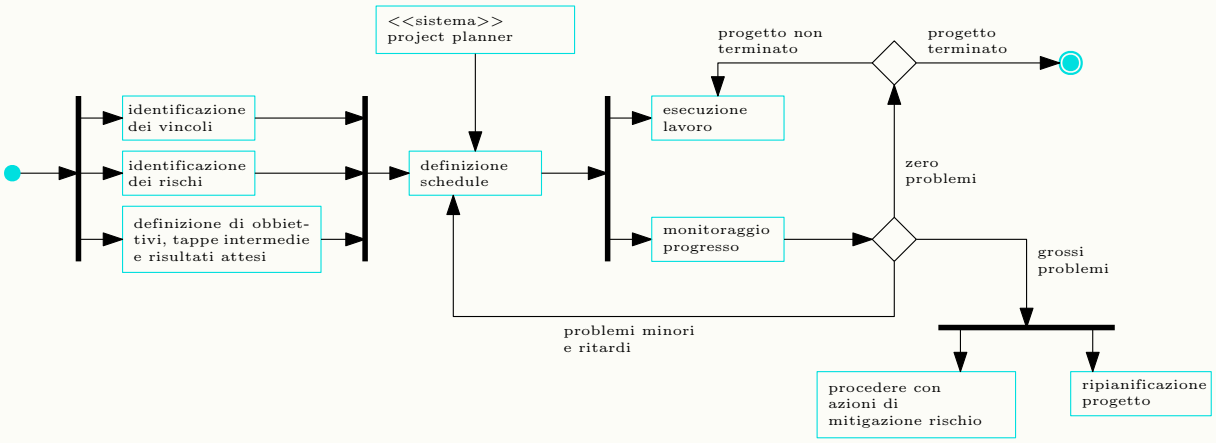


Figura 2.1: Processi del project planning

- Bisogna sempre considerare ogni imprevisto, in modo che eventuali problemi non siano troppo gravanti sulla schedule.

In caso di seri problemi durante lo sviluppo, è necessario applicare delle procedure di *mitigazione del rischio*, onde evitare il fallimento del progetto. In congiunzione a ciò, può essere necessaria la ripianificazione del progetto.
Ciò può comportare la rinegoziazione con il cliente dei risultati attesi e dei vincoli da rispettare. Una nuova schedule, e data di consegna dovrà essere definita in modo da stabilire un accordo con il cliente.

2.2 Scheduling del Progetto

Definizione : : Con *scheduling del progetto* si intende il processo di decisione riguardante il come il carico di lavoro di un progetto deve essere organizzato in differenti attività, ed in che ordine queste attività devono essere eseguite.

Viene stimato un calendario con i tempi necessari al completamento delle attività, lo sforzo necessario ed il personale al quale delegarlo. È anche necessaria una stima delle risorse necessarie, come lo spazio su disco necessario per il software, ed il budget da muovere.

1. Suddivisione del progetto in varie attività e stima delle risorse per ognuna di esse
2. Organizzazione delle attività da eseguire in concorrenza per ottimizzare i tempi
3. Minimizzazione delle dipendenze fra le varie attività, in modo da ridurre i ritardi dovuti ad attese

Questi ultimi fattori dipendono anche dall'intuizione e dalla esperienza del project manager.

Problemi nello scheduling

- Stimare la difficoltà del lavoro ed i costi di sviluppo è molto difficile
- La produttività non è proporzionale al numero di persone coinvolte nel lavoro
- L'aggiunta di personale a progetto già avviato può causare ritardi
- Accade sempre l'inaspettato, bisogna considerare ogni evenienza nel planning

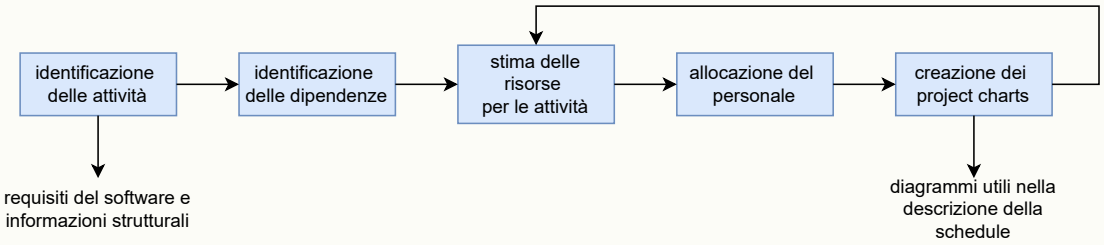


Figura 2.2: Scheduling

Esiste un annotazione grafica utile nella rappresentazione della schedule, mostra le diverse attività, il periodo in cui vanno terminate e le varie dipendenze fra esse, il diagramma a barre mostra le attività come risorse da disporre sull'asse dei tempi. Le "project activities" sono gli elementi di base del grafico, comprendono

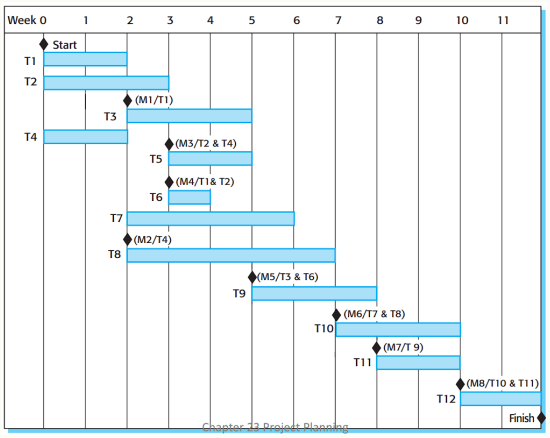
- una durata sul calendario, di giorni o mesi
- un carico di lavoro stimato, misurato in numero di impiegati al giorno necessari
- una deadline per ogni attività che ne vincola il completamento entro una certa data
- un punto specifico che descrive il terminamento di un'attività, può essere un documento , una riunione o il completamento di tutti i test

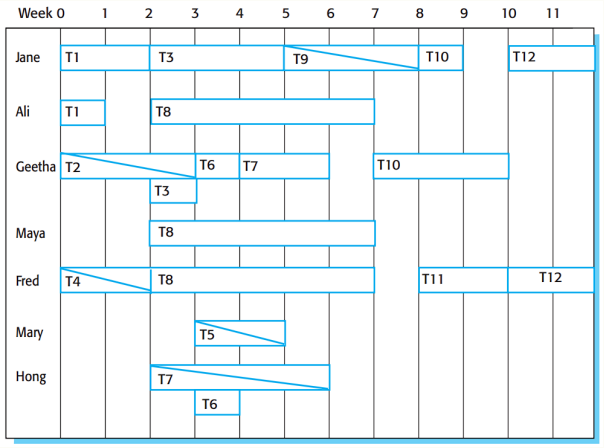
Una **milestone** non è altro che una "tappa fondamentale" durante lo svolgimento delle varie attività, può rappresentare un momento in cui si valuta la progressione del progetto.

Con **deliverables** si definiscono dei risultati ottenuti durante la lavorazione da presentare al cliente.

task	personale necessario (persone/giorno)	durata in giorni	dipendenze
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2,T4 (M3)
T6	10	5	T1,T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10		T3,T6 (M5)
T10	20	15	T7,T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Nella tabella sopramostrata, sono riportate diverse attività di un processo di sviluppo, ogni attività T_i (o task) ha associato un certo numero di persone necessarie ed una durata in giorni. Inoltre, un task può dipendere da uno o più task. Con M_i si identificano le milestones.





Un *activity bar chart* è un diagramma in cui sull'asse temporale vengono rappresentate le varie attività ed il loro scheduling, in modo che le varie dipendenze siano rispettate.

Analogamente, uno *staff allocation chart* mostra la distribuzione del personale suddiviso nelle varie settimane, indicando il progetto alla quale lavorano.

2.3 Sviluppo Agile

Con sviluppo agile, si denota una metodologia di approccio iterativa in cui il processo di sviluppo ed i requisiti vengono rimodellati dinamicamente, in quanto il progetto in fase di sviluppo viene considerato dal cliente prima del suo completamento. Diversamente dallo sviluppo plan driven, questo approccio non è predeterminato e le decisioni vengono prese durante lo sviluppo, possibile aggiunte o rimozioni dipendono da come il progetto procede e dalle nuove priorità del cliente.

> Le priorità e le esigenze del cliente cambiano, quindi ha senso avere un piano flessibile che possa accogliere questi cambiamenti.

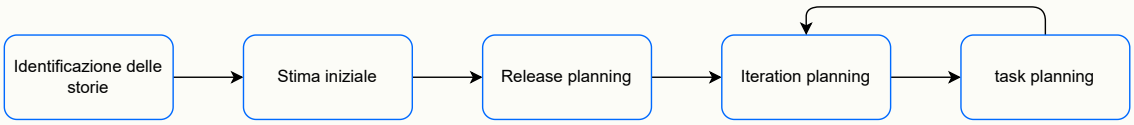
Lo sviluppo agile prevede due stadi

- *Release planning* : prevede la pianificazione a lungo termine delle funzionalità che dovranno essere implementate nei mesi a venire.
- *Iteration planning* : ci si concentra sugli incrementi a breve termine del sistema, tipicamente si tratta di aggiunte che richiedono dalle 2 alle 4 settimane.

Una delle metodologie dello sviluppo agile è la pianificazione basata su storie (**story-based planning**). Consiste nel suddividere le funzionalità di un sistema in piccole unità chiamate "storie utente" (*user stories*). In particolare, sono previsti i seguenti step

1. Creazione delle storie: Le storie utente sono brevi descrizioni di una funzionalità del sistema dal punto di vista dell'utente finale. Solitamente seguono un formato semplice: "Come utente, voglio [azione] in modo da [obiettivo]".
2. Stima dello sforzo: Il team di sviluppo stima il tempo necessario per implementare ciascuna storia, assegnandole un valore numerico (effort points) che riflette la complessità e la dimensione della storia.
3. Prioritizzazione: Le storie vengono ordinate in base alla priorità, tenendo conto di fattori come il valore commerciale, l'urgenza e le dipendenze.
4. Pianificazione delle iterazioni: Le storie prioritarie vengono raggruppate in iterazioni (sprint) di durata definita.
5. Misurazione della velocità: Viene calcolata la "velocità" del team, ovvero la quantità di lavoro (misurata in effort points) che il team riesce a completare in un'iterazione.

6. Previsione: Sulla base della velocità, il team può stimare il tempo totale necessario per completare il progetto.



Il Release Planning si occupa di selezionare e raffinare le storie utente (features) che verranno implementate in una specifica versione (release) del prodotto. Questo processo definisce quali funzionalità saranno incluse nel rilascio e l'ordine in cui verranno sviluppate.

L'Iteration planning si concentra sulla scelta delle storie utente che verranno implementate in ciascuna iterazione (spesso chiamate sprint), che sono periodi di tempo fissi (solitamente 2 o 3 settimane) durante i quali il team si impegna a completare un insieme di lavoro. Il numero di storie scelte per ogni iterazione dipende dalla capacità del team (velocità) di completare il lavoro.

Durante la fase di pianificazione dei task, gli sviluppatori suddividono le storie utente in compiti di sviluppo più piccoli.

- Un task di sviluppo dovrebbe richiedere tra le 4 e le 16 ore.
- Tutti i task necessari per completare tutte le storie di quell'iterazione vengono elencati.
- I singoli sviluppatori si offrono volontari per i task specifici che desiderano implementare.

Benefici di questo approccio:

- L'intero team ha una visione d'insieme dei task da completare in un'iterazione.
- Gli sviluppatori si sentono responsabili dei loro task e questo li motiva a completarli.

Un incremento del software[1] viene sempre consegnato alla fine di ogni iterazione del progetto. Se le funzionalità da includere nell'incremento non possono essere completate nel tempo previsto, si riducono gli obbiettivi prefissati del lavoro, ma la tempistica di consegna non viene mai prolungata.

Svantaggi dell'agile planning

- La pianificazione Agile dipende fortemente dal coinvolgimento e dalla disponibilità del cliente.
- può essere difficile da organizzare, poiché i clienti devono spesso dare priorità ad altro lavoro e non sono sempre disponibili per le sessioni di pianificazione.
- alcuni clienti potrebbero essere più abituati ai piani di progetto tradizionali e trovare difficile partecipare a un processo di pianificazione agile.

La pianificazione agile funziona bene con team di sviluppo piccoli e stabili che possono riunirsi e discutere le storie da implementare. Tuttavia, quando i team sono grandi e/o geograficamente distribuiti, o quando la composizione del team cambia frequentemente, è praticamente impossibile coinvolgere tutti nella pianificazione collaborativa che è essenziale per la gestione agile di un progetto.

> Lo sviluppo agile è un approccio molto efficace per team piccoli e coesi, richiede adattamenti e strumenti specifici per funzionare bene in contesti più grandi e complessi.

[1] un incremento del software rappresenta una porzione funzionante e valutabile del prodotto software che viene consegnata alla fine di ogni iterazione

CAPITOLO

3

PROCESSI SOFTWARE

3.1 Definizione e Modelli

Definizione : Il processo software è un insieme strutturato di attività necessarie per sviluppare un sistema software. Esistono diversi processi software, ma tutti coinvolgono specifiche, progettazione e implementazione, validazione ed evoluzione del sistema per rispondere ai bisogni dei clienti.

Quando descriviamo e discutiamo di processi, di solito parliamo delle attività che li compongono, ad esempio, la specifica di un modello di dati, la progettazione dell'interfaccia utente, ecc., e dell'ordine in cui queste attività vengono eseguite. Le descrizioni dei processi possono includere anche:

- Prodotti: ovvero i risultati di un'attività del processo;
- Ruoli: che riflettono le responsabilità delle persone coinvolte nel processo;
- Pre-condizioni e post-condizioni: ossia affermazioni che sono vere prima e dopo che un'attività del processo è stata eseguita o un prodotto è stato creato.

Come già accennato, i processi software possono essere di due tipi

- **processi plan-driven** : Sono processi in cui tutte le attività vengono pianificate in anticipo e l'avanzamento del progetto viene misurato in base a questo piano iniziale. È come seguire una ricetta precisa, dove ogni passo è definito fin dall'inizio
- **processi agili** : In questi processi, la pianificazione è più flessibile e si adatta ai cambiamenti. Si lavora a iterazioni brevi (sprint) e si può modificare il piano di volta in volta in base alle nuove esigenze del cliente. È come cucinare senza una ricetta precisa, ma adattando i piatti ai gusti degli ospiti man mano che si procede.

Nella pratica La maggior parte dei progetti software combina elementi di entrambi gli approcci. A volte è necessario un piano dettagliato per le fasi iniziali, mentre in altre si può essere più flessibili.

Il processo software vive attraverso differenti formulazioni, o *modelli*, precisamente verranno trattati i seguenti

- *waterfall model*
- *incremental development*
- *integration and configuration*

Nella pratica, i sistemi di grandi dimensioni incorporano elementi da più modelli.

Waterfall Model

È uno dei modelli di sviluppo software più antichi e tradizionali è di tipo plan driven e consiste nell'individuare e separare le distinte fasi di specifica e sviluppo. Tali fasi sono

- analisi e definizione dei requisiti
- progettazione del sistema e del software
- implementazione e collaudo
- integrazione e deployment del sistema
- mantenimento

Gli svantaggi di tale modello risiedono nella poca flessibilità e robustezza nell'accomodare richieste di cambiamento una volta che il progetto è indirizzato su una certa via. Generalmente, è necessario terminare una fase prima di passare alla successiva. Il modello a cascata divide il progetto in fasi distinte e rigide,

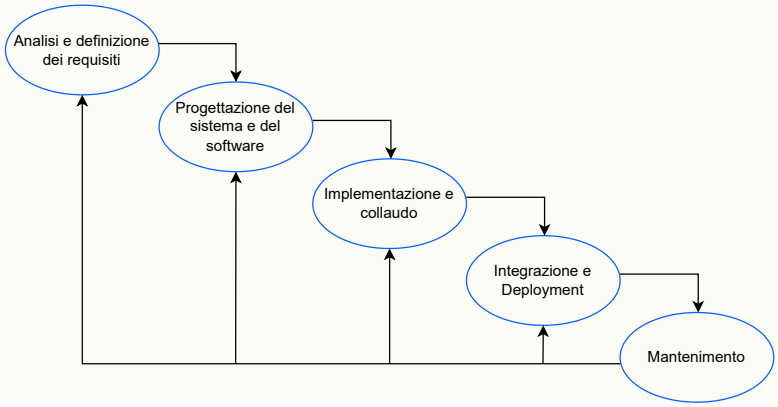


Figura 3.1: Il modello waterfall

rendendo difficile adattarsi ai cambiamenti nei requisiti del cliente.

- Limitazioni nei requisiti: Questo modello è adatto solo quando i requisiti sono ben definiti all'inizio del progetto e non si prevedono grandi modifiche durante lo sviluppo.
- Instabilità dei requisiti: Nella maggior parte dei casi, i requisiti dei sistemi informatici non sono statici, ma cambiano nel tempo.

Il modello a cascata viene spesso utilizzato in progetti di ingegneria software su larga scala, dove il sistema viene sviluppato in più sedi. In questi casi, la natura pianificata del modello a cascata aiuta a coordinare le attività tra i diversi team di sviluppo.

Incremental Development

Tale modello di *sviluppo incrementale* punta alla riduzione dei costi necessari per accomodare le nuove richieste del cliente che insorgono a progetto già avviato. La quantità di analisi e documentazione da rifare è molto inferiore rispetto al modello a cascata : si riduce la necessità di ricominciare da capo se i requisiti cambiano, risparmiando tempo e risorse.

- *È più facile ottenere feedback dal cliente sul lavoro di sviluppo svolto* : Il cliente può vedere e provare il software in fase di sviluppo e fornire un feedback diretto, contribuendo a migliorare il prodotto finale.
- *È possibile una consegna e un deployment più rapidi del software utile per il cliente* : Il cliente può iniziare a utilizzare il software prima rispetto al modello a cascata, ottenendo così un beneficio più rapido dal prodotto.

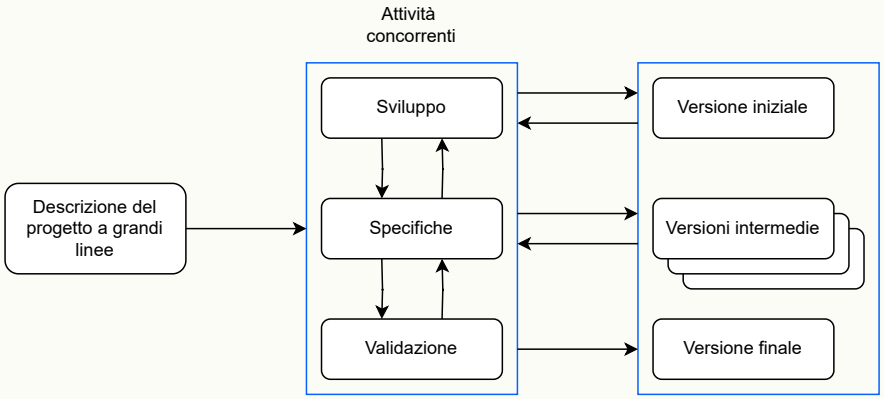


Figura 3.2: Incremental Development

Il modello comporta alcuni svantaggi

- **Mancanza di visibilità per i manager**: I manager spesso necessitano di deliverable regolari per misurare l'avanzamento del progetto. In uno sviluppo incrementale rapido, produrre documentazione dettagliata per ogni versione può risultare costoso e inefficiente. La natura iterativa dello sviluppo incrementale può rendere complesso tracciare tutti i cambiamenti apportati al sistema nel tempo.
- **Degradazione della struttura del sistema**: L'aggiunta continua di nuove funzionalità senza una adeguata manutenzione può portare al degrado della struttura del codice, rendendolo più complesso e meno manutenibile. Man mano che il sistema cresce, diventa sempre più difficile apportare modifiche senza introdurre nuovi bug o impatti negativi su altre parti del sistema.

Integration and configuration

È un modello basato sul riutilizzo del software: I sistemi vengono costruiti combinando componenti software preesistenti o sistemi applicativi già pronti all'uso (spesso chiamati COTS - Commercial-off-the-shelf). Spesso avviene una riconfigurazione degli elementi riutilizzati, tali componenti possono essere personalizzati per adattarsi alle specifiche esigenze dell'utente.

Il riutilizzo del software è diventato il metodo più comune per sviluppare molti tipi di sistemi aziendali, verrà trattato approfonditamente in seguito. Esistono diversi tipi di **software riutilizzabile**

- Sistemi applicativi autonomi (a volte chiamati COTS) che vengono configurati per l'uso in un ambiente specifico.
- Collezioni di oggetti sviluppati a mò di pacchetto da integrare in un framework di componenti come .NET o J2EE.
- Servizi web sviluppati secondo standard e disponibili per essere integrati remotamente.

Le fasi chiave del processo sono

1. Specifica dei requisiti
2. Definizione e valutazione del software
3. Raffinamento dei requisiti
4. Configurazione del sistema
5. Adattamento e integrazione dei componenti

Vantaggi

- Costi e rischi ridotti, poiché si sviluppa meno software da zero

- Consegna e implementazione più rapida del sistema

Svantaggi

- sono inevitabili dei compromessi sui requisiti, quindi il sistema potrebbe non soddisfare pienamente le esigenze degli utenti
- Perdita di controllo sull'evoluzione degli elementi del sistema riutilizzati, Non avendo il controllo completo sul codice sorgente dei componenti riutilizzati

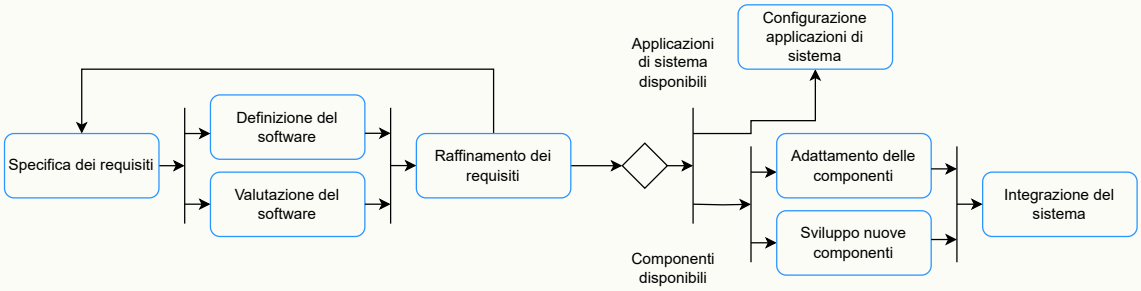
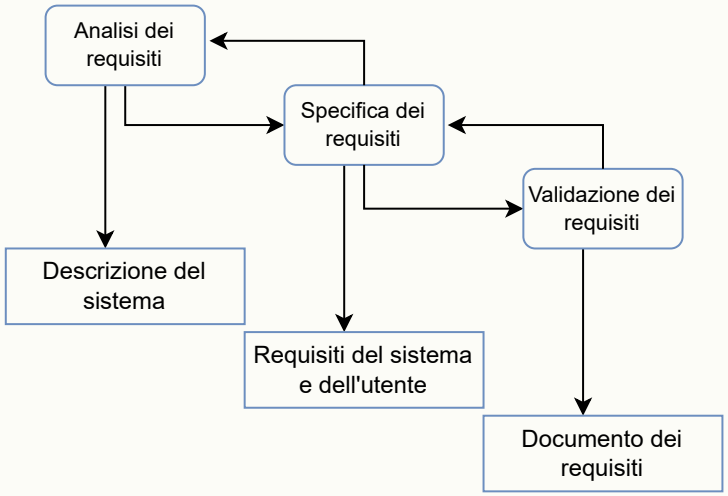


Figura 3.3: Ingegneria del software orientata al riciclo

È importante anche l'ingegneria dei requisiti, descritta dal seguente diagramma



Specifica del software

Con "processo di definizione" si denota il processo in cui si stabilisce con precisione quali servizi deve fornire il software e quali sono i vincoli che ne limitano il funzionamento e lo sviluppo. Si definisce l'ingegneria dei requisiti come

- Estrazione e analisi: Si identificano e si analizzano le esigenze e le aspettative di tutti coloro che interagiscono con il sistema (stakeholder).
- Specifiche dei requisiti: Si definiscono nel dettaglio i requisiti individuati.
- Validazione dei requisiti: Si verifica che i requisiti siano corretti, completi e coerenti.

I passi della progettazione ed implementazione del software sono

- Conversione in sistema eseguibile: Si trasforma la specifica in un programma funzionante.
- Progettazione del software: Si crea una struttura del software che soddisfi i requisiti definiti nella specifica.
- Implementazione: Si traduce la struttura progettata in un programma eseguibile (codice).
- Interrelazione: Le attività di progettazione e implementazione sono strettamente correlate e possono essere svolte in modo iterativo.

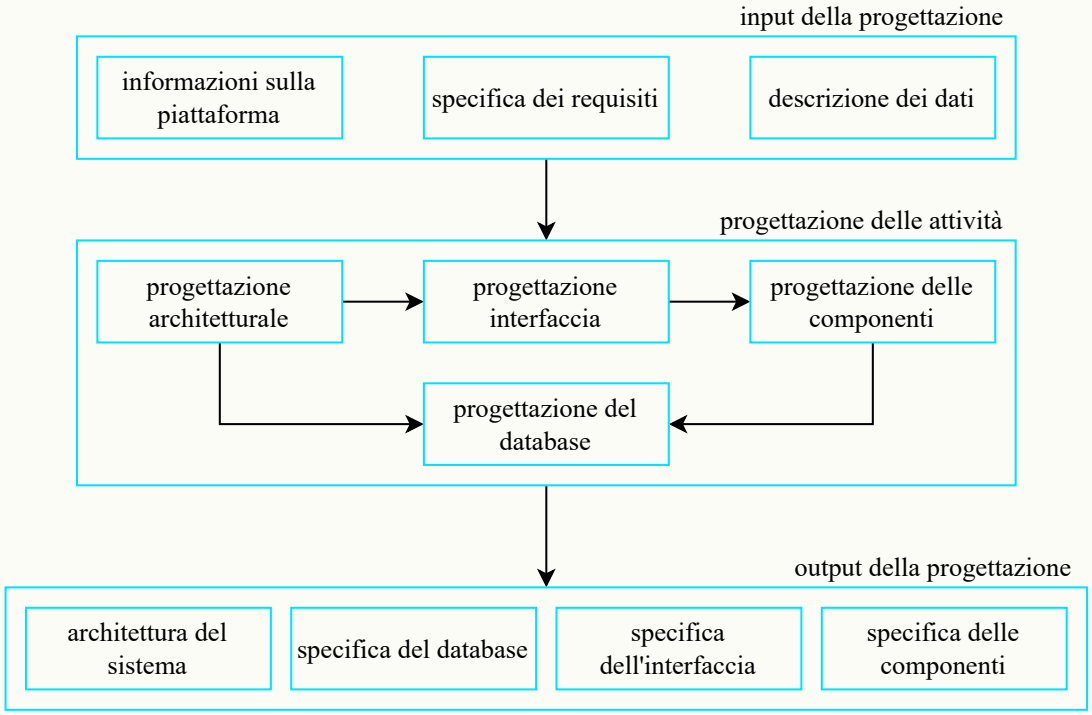
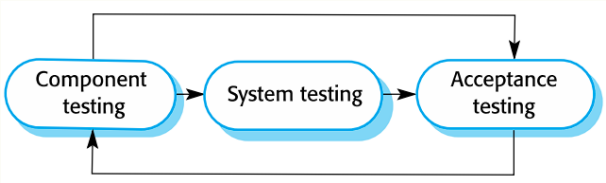
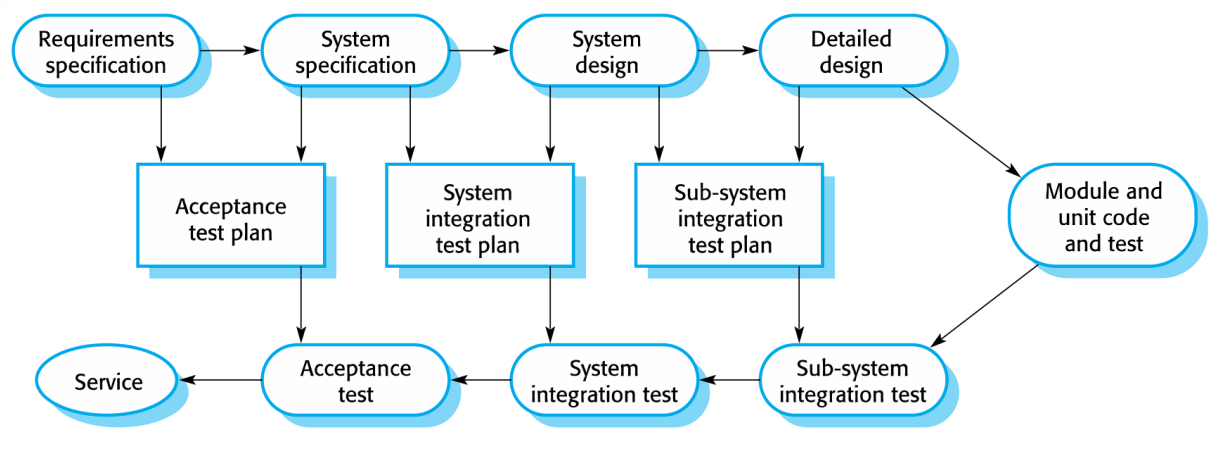


Figura 3.4: modello generale del processo di progettazione

Le componenti del sistema passano poi per una serie di fasi di testing



- Testing dei componenti : I singoli componenti vengono testati in modo indipendente. I componenti possono essere funzioni, oggetti o gruppi coerenti di queste entità.
- Testing di sistema: Il sistema viene testato nella sua interezza. Il testing delle proprietà emergenti è particolarmente importante.
- Testing del cliente: Il sistema viene testato utilizzando i dati del cliente per verificare che soddisfi le esigenze del cliente.


Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and test
Service
Acceptance test
System integration test
Sub-system integration test

CAPITOLO

4

SVILUPPO AGILE

Uno dei requisiti più importanti nell'ambito dei sistemi software odierni, è la rapidità nello sviluppo e nel deployment del servizio, il business in questione spesso evolve rapidamente ed è praticamente impossibile definire un insieme di requisiti stabili nel tempo. Il software, deve evolvere rapidamente a sua volta per stare dietro ai requisiti mutevoli del business.

Lo sviluppo *plan driven* è essenziale per alcuni tipi di software, ma non riesce a stare al passo con i bisogni del business. Lo *sviluppo agile*, nato alla fine degli anni 90' del secolo scorso, punta a ridurre drasticamente i tempi di consegna, fornendo rapidamente del software funzionante e pronto all'utilizzo.

I concetti chiave dello sviluppo agile sono i seguenti

- **Interleaving delle fasi**: La specifica, la progettazione e l'implementazione del programma sono intrecciate tra loro. Ciò significa che queste fasi non seguono una sequenza rigida, ma si sovrappongono e si influenzano a vicenda durante tutto il processo di sviluppo.
- **Sviluppo incrementale**: Il sistema viene sviluppato in modo graduale, attraverso una serie di versioni o incrementi. Ad ogni incremento vengono aggiunte nuove funzionalità o migliorate quelle esistenti, coinvolgendo sempre gli stakeholder nella definizione e nella valutazione di ogni versione.
- **Consegne frequenti**: Nuove versioni del software vengono consegnate frequentemente per ottenere un feedback tempestivo dagli utenti e dagli stakeholder. Questo permette di apportare modifiche e miglioramenti in modo continuo.
- **Ampio supporto degli strumenti**: Lo sviluppo agile si avvale di numerosi strumenti, come ad esempio gli strumenti di test automatizzati, per supportare il processo di sviluppo e garantire la qualità del software.
- **Documentazione minima**: L'enfasi è posta sul codice funzionante piuttosto che su una documentazione estesa. La documentazione viene creata solo quando è strettamente necessaria.

Lo sviluppo plan driven (o plan based) è caratterizzato da

- Approccio sequenziale: In questo modello, le fasi di sviluppo (specifica, progettazione, implementazione, test) sono ben definite e seguono un ordine preciso, come in un piano.
- Pianificazione anticipata: I risultati attesi da ciascuna fase sono pianificati in anticipo, creando una roadmap dettagliata del progetto.
- Non necessariamente a cascata: Anche se spesso associato al modello a cascata, lo sviluppo guidato da un piano può essere incrementale, ovvero suddiviso in fasi più piccole. Tuttavia, la pianificazione generale rimane rigida.

- Iterazioni interne: Le iterazioni avvengono all'interno di ciascuna attività, ma il flusso generale del progetto rimane lineare.

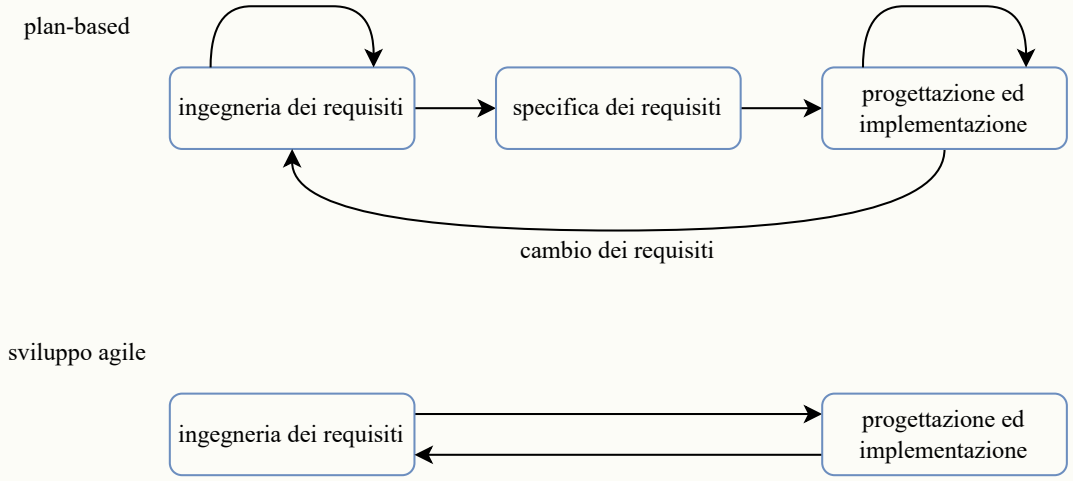


Figura 4.1: differenze fra plan-driven e sviluppo agile

4.1 Agile Methods

- La nascita dei metodi agili è stata motivata dalla frustrazione nei confronti dei metodi di progettazione software degli anni '80 e '90, considerati troppo rigidi e burocratizzati.
- Questi, pongono l'accento sul codice funzionante piuttosto che su una documentazione eccessiva. L'idea è che il codice costituisca la migliore forma di documentazione.
- Lo sviluppo avviene attraverso cicli iterativi, dove si costruiscono e si migliorano gradualmente le funzionalità del software.
- I metodi agili sono progettati per adattarsi rapidamente ai cambiamenti dei requisiti, che sono inevitabili in molti progetti software.

L'obiettivo principale dei metodi agili è ridurre al minimo gli sprechi e le inefficienze nel processo di sviluppo software. Ciò si ottiene limitando la documentazione non necessaria e focalizzandosi sulla consegna rapida di software funzionante. Inoltre, i metodi agili permettono di rispondere in modo flessibile ai cambiamenti dei requisiti, evitando di dover rifare grandi porzioni di lavoro.

L' **Agile Manifesto** è una dichiarazione di principi che ha dato origine ai metodi agili di sviluppo software. Quest'ultimo recita

> *Stiamo scoprendo metodologie migliori di produrre software tramite la produzione di software stessa e l'aiuto reciproco nel farlo. Attraverso questo lavoro, abbiamo imparato ad apprezzare*
>
> - Individui e interazioni più che processi e strumenti
> - Software funzionante più che documentazione esaustiva
> - Collaborazione con il cliente più che negoziazione contrattuale
> - Rispondere al cambiamento più che seguire un piano

Volendo riassumere il tutto, è possibile identificare i seguenti principi dello sviluppo agile:

Principio	Descrizione
coinvolgimento del cliente	il cliente dovrebbe essere strettamente coinvolto nel processo, il suo ruolo è quello di fornire nuovi requisiti e di valutare le iterazioni del sistema
consegna incrementale	Il software viene sviluppato in incrementi, con il cliente che specifica i requisiti da includere in ciascun incremento
priorità alle persone, non ai processi	Le competenze del team di sviluppo dovrebbero essere riconosciute e sfruttate. I membri del team dovrebbero essere liberi di sviluppare i propri modi di lavorare senza processi prescrittivi
accogliere il cambiamento	Prevedere che i requisiti del sistema cambieranno e quindi progettare il sistema per accogliere eventuali cambiamenti
Mantenere la semplicità	Focalizzarsi sulla semplicità sia nel software che si sta sviluppando che nel processo di sviluppo. Laddove possibile, lavorare attivamente per eliminare la complessità dal sistema

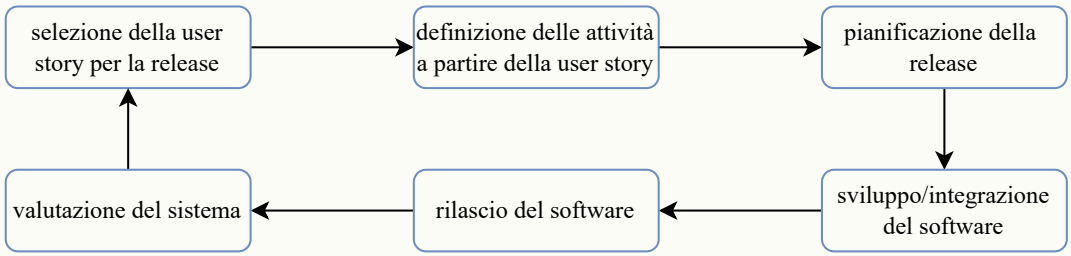
Riguardo l'**applicabilità** dei metodi agili, queste metodologie sono adatte a software di piccole o medie dimensioni, destinati alla vendita. È inoltre efficacie nell'ambito dello sviluppo di sistemi personalizzati all'interno di un'organizzazione: I metodi agili sono ideali quando il cliente è fortemente coinvolto nel processo di sviluppo e quando ci sono poche restrizioni esterne (regole, normative) che influenzano il software.

4.2 Tecniche di Sviluppo Agile

4.2.1 Extreme Programming

La prima tecnica attuata nello sviluppo agile che verrà considerata è l'*extreme programming* (XP), è stata una metodologia molto influente sviluppata alla fine degli anni '90. XP ha introdotto una serie di tecniche agili che hanno fortemente influenzato il modo in cui pensiamo allo sviluppo software. Risulta essere un approccio "estremo" allo sviluppo iterativo: XP porta all'estremo i principi dello sviluppo iterativo, ovvero la creazione di versioni incrementali del software a intervalli regolari. Le caratteristiche principali sono

- Frequenti release: Nuove versioni del software possono essere create anche più volte al giorno
- Consegne regolari: Incrementi di funzionalità vengono consegnati ai clienti ogni 2 settimane circa.
- Test continui: Tutti i test automatici vengono eseguiti ad ogni nuova build del software e la build viene accettata solo se tutti i test superano con successo.



I principi della XP sono i seguenti

1. Pianificazione incrementale : I requisiti vengono registrati su "story cards" e le storie da includere in un rilascio vengono determinate in base al tempo disponibile e alla loro priorità relativa. Gli sviluppatori suddividono queste storie in "Task" di sviluppo.

2. Rilasci frequenti : Viene sviluppato prima l'insieme minimo di funzionalità utili che fornisce valore al business. I rilasci del sistema sono frequenti e aggiungono in modo incrementale funzionalità al primo rilascio.
3. Design semplice : Viene effettuata solo la progettazione necessaria per soddisfare i requisiti attuali.
4. Sviluppo guidato dai test : Viene utilizzato un framework di test unitari automatizzato per scrivere i test per una nuova funzionalità prima che questa venga implementata.
5. Refactoring : Tutti gli sviluppatori sono tenuti a ristrutturare continuamente il codice non appena vengono trovati possibili miglioramenti. Ciò mantiene il codice semplice e manutenibile.
6. Programmazione "a coppie" : Gli sviluppatori lavorano in coppia, controllando reciprocamente il lavoro e fornendo supporto per garantire sempre un buon risultato.
7. Proprietà collettiva del codice : Le coppie di sviluppatori lavorano su tutte le aree del sistema, evitando la creazione di "isole di competenza" e garantendo che tutti siano responsabili del codice. Chiunque può apportare modifiche.
8. Integrazione continua : Non appena un compito è completato, viene integrato nel sistema completo. Dopo ogni integrazione, tutti i test unitari devono essere superati.
9. Ritmo sostenibile : Non sono ammessi eccessivi straordinari, poiché riducono la qualità del codice e la produttività a lungo termine.
10. Cliente in loco : Un rappresentante dell'utente finale deve essere sempre disponibile per il team XP. In XP, il cliente è parte integrante del team e ha la responsabilità di portare i requisiti del sistema al team per l'implementazione.

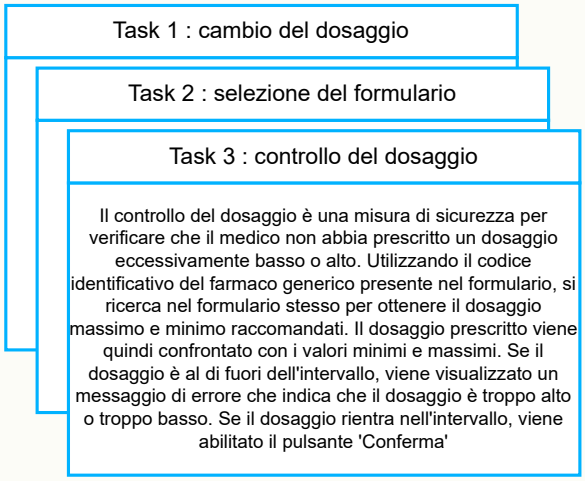
L'XP ha definito alcuni pratiche molto influenti nell'ambito dell'ingegneria del software, l'extreme programming ha un carattere molto tecnico, ed è difficile da integrare nelle pratiche gestionali di molte organizzazioni, a tal proposito, sebbene lo sviluppo agile si basi su molte pratiche di XP, il metodo XP nella sua forma originale non è ampiamente utilizzato. Le pratiche più importanti che si sono diffuse sono

- User stories per la specifica
- Refactoring
- Sviluppo guidato dai test
- Programmazione "a coppie"

Nel capitolo 2 si è già accennato il concetto di **storie utente**. I requisiti sono espressi come "storie utente" o scenari, e tali storie vengono suddivise in "task" più piccoli, che servono come base per la pianificazione. Il cliente decide quali storie includere nel prossimo rilascio in base alla priorità e alle stime temporali.

Prescrizione della Medicazione
Il record del paziente deve essere aperto per l'inserimento. Clicca sul campo "medicazione" e seleziona tra "medicazione attuale", "nuova medicazione" o "formulario". Se selezioni "medicazione attuale", ti verrà chiesto di verificare il dosaggio; se desideri modificarlo, inserisci il nuovo dosaggio e conferma la prescrizione. Se scegli "nuova medicazione", il sistema assumerà che tu conosca il farmaco da prescrivere. Digita le prime lettere del nome del farmaco. Ti verrà mostrato un elenco di possibili farmaci. Scegli quello corretto. Ti verrà quindi chiesto di confermare il farmaco selezionato. Inserisci il dosaggio e conferma la prescrizione. Se scegli "formulario", ti verrà presentata una casella di ricerca per il formulario approvato. Cerca il farmaco desiderato e selezionalo. Ti verrà quindi chiesto di confermare il farmaco selezionato. Inserisci il dosaggio e conferma la prescrizione. In tutti i casi, il sistema verificherà che il dosaggio sia all'interno dell'intervallo consentito e ti chiederà di modificarlo se è al di fuori dei dosaggi raccomandati. Dopo aver confermato la prescrizione, questa verrà visualizzata per la verifica. Clicca su "OK" o "Modifica". Se clicchi su "OK", la prescrizione verrà registrata nel database delle audit. Se clicchi su "Modifica", ricomincerai il processo di "Prescrizione del farmaco".

Dall'esempio appena mostrato di storia utente, possono essere derivate più task.



Continua da "examples of refactoring" pagina 5 slide di Marco Maz.