CN6

Obiettivo Qualità verso la fabbrica del software affidabile

P. Nesi

Milano
11 - 12 - 13 maggio 1994

Centro per le Tecnologie Informatiche CARLO GHIGLIENO

asic

Come trasformare la necessità di ottenere la certificazione ISO9000 in una opportunità per iniziare un processo di miglioramento progressivo e costante della qualità dei prodotti software?
Come impostare l'attività del testing applicandola fin dalle prime fasi del ciclo di vita del software?
Come utilizzare l'attività di testing per monitorare la qualità del processo produttivo?
Cosa conviene automatizzare e cosa non conviene automatizzare nel ciclo di produzione del software?
Quali metriche e quali tools adottare?

A queste domande si propone di dare una risposta il convegno sulla qualità nato dalla collaborazione fra il Centro per le Tecnologie Informatiche Carlo Ghiglieno e ASIC S.r.l. con il coordinamento scientifico dell'Ing. A. Serra.

La vertiginosa crescita funzionale dei prodotti software dell'ultima generazione ha fortemente incrementato la complessità stessa delle soluzioni applicative per l'utente, creando presupposti potenziali per un software ad alto rischio di malfunzionamenti ed errori.
È diventato pertanto essenziale, per le società di software, raggiungere un livello di maturità organizzativa tale da sviluppare prodotti a qualità costante in modo indipendente, per quanto possibile, dalle funzionalità richieste e dalle capacità dei suoi sviluppatori.
Il convegno vuole fornire elementi sia a livello tecnico che manageriale, utili ad orientarsi in quell'autentico percorso ad ostacoli, come ha detto qualcuno, a metà tra tecnologia ed organizzazione.

Una Survey sullo stato dell'arte delle tecniche di testing verrà presentata dal Dr. Edward Miller, presidente della Software Research Inc., società specializzata nello sviluppo di tools per il software testing.
Edward Miller, è uno dei massimi esperti sulla qualità del software. È stato coinvolto nello sviluppo di numerosi tools per l'automazione del software testing e software analysis. È stato chairman del *Ist International Conference on Computer Workstations* nel 1985. Ha partecipato per diversi anni alla organizzazione di conferenze presso l'IEEE ed è l'autore di *Software Testing and Validation Techniques,* pubblicazione della *IEEE Computer Society Press.*

Chairman:
Ing. A. Serra (ASIC S.r.l.)

Relatori:
Ing. V. Asnaghi (Laboratorio Software IMQ)
Prof. G. Bruno (Dip. di Automatica ed Informatica del Politecnico di Torino)
Ing. M. Campanai (CQ_ware, Centro per la Qualità del Software)
Dr. A. Cicu (Bull Italia, Advanced Research Department)
Ing. A. Di Maio (TXT Ingegneria Informatica S.p.A.)
Dr. P. Mariani (Società Italiana Avionica)
Ing. A. Martelli (Alenia Spazio)
Dr. E. Miller (Software Research Inc.)
Dr. Ing. P. Nesi (Dip. di Sistemi ed Informatica dell'Università di Firenze)
Ing. L. Sario (Alenia Spazio)
Ing. M. Schettini (Olivetti Ricerca)
Ing. A. Serra (ASIC S.r.l.)

È prevista la traduzione simultanea.

Paradigma Orientato agli Oggetti nella Specifica dei Sistemi di Tempo Reale

Paolo Nesi
Dipartimento di Sistemi e Informatica, Facoltà di Ingegneria
Università degli Studi di Firenze, Via S. Marta 3, 50139 Firenze, Italy
tel.: 055-4796265, fax.: 055-4796363, nesi@ingfi1.ing.unifi.it

Abstract

Le tecniche per la specifica di sistemi di tempo reale sono attualmente al centro dell'attenzione di molti ricercatori e sviluppatori. Questo è dovuto al fatto che i sistemi di tempo reale sviluppati per mezzo di tecniche tradizionali presentano usualmente forti criticità dal punto di vista dell'affidabilità a causa dell' adozione di metodologie di sviuluppo idadegute e di metodi empirici per la soluzione di problemi specifici. Negli ultimi anni il paradigma orientato agli oggetti è stato utilizzato da molti per migliorare il processo di produzione software dei sistemi di tempo reale, definendo metodologie, tool e linguaggi. In molti casi, il paradigma è stato affiancato da metodi formali per garantire la possibilità di descrivere tutti gli aspetti del sistema ed effettuare verifiche di consistenza e completezza della specifica, nonché la validazione finale. Questa deve garantire che il sistema finale soddisferà i vincoli temporali imposti sul comportamento del sistema dal contesto. In questo lavoro sono evidenziate le caratteristiche che deve avere un metodo per lo sviluppo di software per sistemi di tempo reale al fine di assicurare il corretto funzionamento del sistema finale ed un'elevata qualità con uno sforzo ragionevole.

1 Introduzione

Negli ultimi anni sono state proposte molte nuove tecniche per la specifica dei sistemi di tempo reale. Il crescente interesse per queste nuove tecniche è dovuto al fatto che molte di queste sono in grado di garantire una maggiore qualità del software prodotto e una maggiore produttività, fornendo la possibilità di eseguire verifiche formali di consistenza, completezza, etc.

Secondo una classificazione molto diffusa, le tecniche di specifica si possono catalogare sulla base del loro grado di formalismo. Vengono detti metodi *informali* quelli che *non hanno* una sintassi e una semantica rigorosa (e.g., il linguaggio naturale). I metodi *formali* sono definiti specificando matematicamente sia la sintassi che la semantica (quest'ultima può essere definita anche solo in parte). Per la specifica dei sistemi software i metodi formali sono di gran lunga preferiti a quelli informali poiché permettono di effettuare delle verifiche di consistenza e completezza [44], [63], [64]. I metodi formali possono essere classificati in *descrittivi*, *operazionali* e *duali*. Gli *operazionali* sono per definizione quelli la cui semantica può essere ricondotta ad una macchina a stati, e pertanto sono intrinsecamente eseguibili. I metodi *descrittivi* si fondano su basi matematiche come assiomi, clausole, asserzioni, etc., questi descrivono il comportamento del sistema in modo astratto per mezzo di espressioni logiche o algebriche. A differenza dei metodi operazionali la semantica dei metodi descrittivi non è direttamente eseguibile. D'altra parte sulla base di tali metodi si può

verificare matematicamente se il sistema specificato soddisfa determinate proprietà di alto livello. I metodi *duali* sono stati definiti cercando di integrare questi due approcci, al fine di avere un unico metodo con il quale fornire la possibilità di eseguire delle verifiche formali su base matematica e di eseguire la specifica.

A prescindere dal tipo di approccio, al fine di poter specificare in modo esaustivo un'applicazione si devono poter modellare gli aspetti strutturali, funzionali e comportamentali [30], [74]. L'aspetto *strutturale* si riferisce a come il sistema viene decomposto in sottosistemi; quello *funzionale* a come sono trasformati i dati all'interno del sistema (per mezzo di algoritmi per la trasformazione dati); il *comportamentale* si riferisce a come il sistema reagisce agli stimoli esterni e si comporta per generare quelli interni (sincroni o asincroni che siano). Sistemi per i quali la risposta ad eventi esterni e molto importante si dicono sistemi reattivi, quelli dove si hanno vincoli stringenti sui tempi di risposta e su quelli di esecuzione in genere, sono detti di tempo reale. I metodi operazionali sono particolarmente adatti a descrivere nel dettaglio il comportamento del sistema dal punto di vista operativo, proprio per questo non consentono la verifica formale (per mezzo di prove di proprietà) del comportamento del sistema come i descrittivi. D'altra parte gli operazionali sono adatti a supportare l'esecuzione diretta e quindi la simulazione della specifica. I metodi descrittivi non sono in grado di descrivere gli aspetti strutturali e funzionali del sistema, mentre sono in grado di descrivere il comportamento del sistema solo in modo astratto. Pertanto, tali metodi non sono intrinsecamente eseguibili poiché la loro esecuzione consiste nella verifica delle proprietà di alto livello del sistema e tale verifica viene eseguita per mezzo di motori inferenziali (e.g., prolog, provatori, etc.). Il processo inferenziale difficilmente può tenere conto dell'ordine sequenziale con il quale devono essere eseguite le operazioni, pertanto il processo di prova di una proprietà non ha niente a che vedere con l'evoluzione temporale dello stato del sistema che invece è direttamente specificata dai metodi operazionali; che sono per questo intrinsecamente eseguibili.

In ogni caso, sia che si adotti un metodo descrittivo o operazionale, il ciclo di vita di un sistema software può essere schematizzato in fasi: collezionamento dei requisiti, analisi dei requisiti (analisi), progetto astratto (decomposizione/composizione), progetto dettagliato (codifica), test (verifica e validazione), mantenimento (manutenzione). È ormai un concetto acquisito che il processo di produzione del software deve essere supportato da una opportuna metodologia; questa deve essere sufficientemente formale da poter assicurare la qualità del prodotto sotto sviluppo. La presenza di una chiara metodologia assicura la ripetibilità delle fasi che portano alla definizione del sistema, rendendo più indipendente la qualità dello sviluppo dallo stile dell'analista. La metodologia deve essere in grado di guidare l'analista/sviluppatore nel definire la gerarchia strutturale del sistema, possibilmente supportando sia uno sviluppo *bottom-up* che *top-down*, favorendo il riuso delle specifiche e/o dei componenti software sviluppati in precedenza. Per poter supportare a pieno il riuso il modello deve essere in grado di descrivere gli aspetti strutturali in modo formale, infatti, molti metodi descrittivi (non avendo un supporto per la definizione della struttura del sistema) non permettono il riuso né delle specifiche complete né di loro parti. In generale, si deve notare che per la loro tendenza all'astrattezza i metodi descrittivi sono più adatti a coprire le fasi in cui si collezionano e si analizzano i requisiti, mentre gli approcci operazionali sono più adatti a coprire la fasi di progetto astratto e codifica. Recentemente, sulla base del paradigma di programmazione ad oggetti (Object-Oriented Paradigm, OOP), sono state definite metodologie e linguaggi particolarmente adatti al riuso anche di specifiche di sistemi di tempo reale. Questo è stato possibile grazie ai meccanismi stessi del paradigma (e.g., inheritance, instantiation, etc.) [48], [8], [46].

Da questa panoramica si deduce che entrambe gli approcci devono essere integrati per poter

coprire tutte le fasi del ciclo di vita del software di sistemi di tempo reale. L'integrazione può essere effettuata a livello di approccio o di linguaggio, come nel caso di metodi duali, oppure integrando l'approccio con altri metodi che coprono gli altri apsetti. Si vengono pertanto a creare dei veri e propri CASE (Computer Aid Software Engineering) tool per la specifica, dotati di interfaccia visuale amichevole, supportati da generatori di codice, etc. Una trattazione più estesa sui tool per la specifica di sistemi di tempo reale può essere reperita in [12].

Al fine di presentare una visione panoramica dei problemi della specifica di sistemi di tempo reale nella sessione 2 è riportato un breve sommario. Nella sessione 3 sono presentati i concetti dai quali il paradigma orientato agli oggetti ha preso spunto e coi quali può essere confrontato. Nella sessione 4 sono esposti i concetti del paradigma e le sue criticità nel modellare i sistemi di tempo reale. In Sect.5 è riportata una breve panoramica sugli approcci ad oggetti per la specifica dei sistemi di tempo reale, questa e seguita da una discussione sulle metodologie ad oggetti, sul ciclo i vita, e sulla gestione dello sviluppo di applicazioni orientate agli oggetti. Alcune considerazioni di carattere generale sono riportate nella sessione 6.

2 Sistemi di Tempo Reale

Le tecniche per la specifica di sistemi di tempo reale (da molto tempo utilizzate in campo militare e per il controllo di macchine automatiche complesse) sono attualmente al centro dell'attenzione di molti ricercatori e sviluppatori. Questo è dovuto alle aumentate necessità di prestazioni che hanno trasformato molti sistemi informativi in sistemi di tempo reale – e.g., applicazioni distribuite, interfacce utente, etc. Nella specifica di sistemi di tempo reale si hanno tutti problemi che sono presenti nella specifica di sistemi senza vincoli temporali, in più il comportamento del sistema dipende dal tempo. Pertanto, anche l'approccio di specifica deve prevedere particolari costrutti per (i) definire i requisiti temporali del sistema, (ii) garantire che il sistema finale soddisferà i vincoli temporali che vengono definiti durante la fase di analisi.

Nell'analisi di sistemi di tempo reale deve essere fatta particolare attenzione all'identificazione dei requisiti temporali del sistema. Questi requisiti possono essere formalizzati associando a certi eventi dei vincoli temporali (*temporal constraints*), per esempio: *timeout* (attesa di un evento con scadenza temporizzata), *deadline* (tempo entro il quale una certa operazione deve essere completata), etc. [63]. Nei vincoli o nelle relazioni temporali il tempo può essere esplicito o implicito. Se il tempo è esplicito può essere espresso in forma relativa o assoluta. Nel primo caso, le durate (e.g., tempi di esecuzione, comunicazione, etc.) e le deadline sono espresse in unità virtuali di tempo. In questo caso la relazione fra unità di tempo e il tempo reale non è chiara. Questo rende più indipendente la verifica dalle caratteristiche hardware del sistema finale ma non vi è garanzia che nel sistema finale saranno soddisfatti i vincoli temporali. Quando il tempo è dato in forma assoluta ci si riferisce al tempo in milli-, micro- o secondi, etc., pertanto l'approccio deve tenere conto delle caratteristiche del sistema finale (tipo di macchina, numero di processi, etc.) per produrre risultati di un qualche senso. Se implicito, il metodo è in grado di esprimere solamente le relazioni temporali fra eventi in termini di precedenze (e.g., l'evento A avviene dopo l'evento B). In questo caso la possibilità di imporre vincoli temporali stretti legati a misure assolute di tempo è impossibile (e.g., entro 5 sec.). L'uso dei vari modelli deve essere valutato in base alle necessità specifiche, se per esempio sono note le caratteristiche hardware del sistema e i vincoli temporali sono molto stringenti è preferibile avere un modello esplicito ed assoluto del tempo.

L'approccio per la specifica deve supportare a livello semantico il concetto di tempo e di vincolo temporale; e quindi il comportamento atteso del sistema (vincoli temporali compresi), deve essere verificato analizzando la specifica del sistema al fine di garantire un comportamento deterministico del sistema. Le verifiche di completezza e correttezza vengono usualmente effettuate in modo statico, sulla base della sintassi e la semantica del metodo di specifica. Dopo tali verifiche deve essere effettuata la validazione della specifica, questa consiste nel controllare che il comportamento del sistema è privo di condizioni di stallo (*deadlock free*), è sicuro (*safe*), e che i vincoli temporali sono rispettati anche al momento dell'esecuzione. Il processo di validazione viene usualmente effettuato in modo statico in approcci descrittivi (per mezzo di prove di proprietà), mentre in quelli operazionali viene effettuato per mezzo di simulazioni.

Per garantire il corretto funzionamento del sistema finale, la semantica dell'approccio deve essere sufficientemente ben definita da consentire di verificare nelle varie fasi dello sviluppo il comportamento del sistema in modo da garantire la predicibilità del comportamento del sistema durante l'esecuzione finale (*run-time*). A questo fine, il supporto per la specifica (linguaggio o tool) ed il sistema operativo sono spesso strettamente vincolati l'un l'altro, per consentire al supporto di tenere conto delle risorse del sistema, intese come: potenza della CPU, timer hardware, memoria cache, canali di comunicazione, etc. Pertanto, certi sistemi basano la validazione finale sulla presenza di un particolare *scheduler* o di un particolare sistema operativo di tempo reale [28], [68]. La maggior parte degli scheduler sono stati definiti ipotizzando che i processi e gli eventi che raggiungono il sistema (in generale classificabili come periodici e aperiodici), sono periodici con periodo e tempo di esecuzione noto. Queste ipotesi consistono in una grossa limitazione quanto si specificano sistemi di tempo reale ma permettono di definire scheduler in grado di garantire un'elevata predicibilità sul comportamento del sistema finale. Un'alternativa è la gestione dinamica dei processi agendo direttamente sulle priorità dei processi. Questa tecnica può creare grossi problemi, come quello dell'*inversione di priorità* per il quale processi a bassa priorità possono in certi condizioni critiche assumere priorità anche maggiori dei processi ad alta priorità del sistema, e quindi portare a delle incongruenze.

Si noti che nelle discussioni precedenti è stato dato per scontato che il sistema viene modellato come un insieme di processi concorrenti che comunicano scambiandosi messaggi con opportuni protocolli. I protocolli possono prevedere sia comunicazioni sincrone (per garantire le sincronizzazioni fra processi) che asincrone. I meccanismi *sincroni* sono più predicibili ma anche più sensibili alla formazione di condizioni di stallo, mentre le comunicazioni *asincrone* sono meno predicibili e meno sensibili alla formazione di condizioni di stallo. In generale, le comunicazioni sincrone sono più adatte per la specifica di sistemi di tempo reale poiché in questi la predicibilità è di primaria importanza, e che le condizioni di stallo devono poter essere individuate durante la fase di validazione della specifica.

Vi sono anche approcci dove la predicibilità non è garantita in senso stretto e quindi la validazione finale viene effettuata solo tramite simulazioni reiterate in varie condizioni di lavoro. In questi metodi, quasi sempre operazionali (e.g., SDL [54]), viene data la possibilità di definire delle procedure per il recupero di condizioni di allarme (i.e., *recovering from failure*), queste possono essere *timeout* perduti, *deadline* non rispettate, singolarità nei calcoli, etc.

In alcuni metodi, al fine di garantire la predicibilità e quindi un assoluto determinismo riguardo al comportamento sistema, vengono imposte delle restrizioni – e.g., assenza di creazione dinamica di processi, assenza di allocazione dinamica di dati, assenza di cambi di priorità, assenza di recurzione, assenza di loop illimitati, etc. [59].

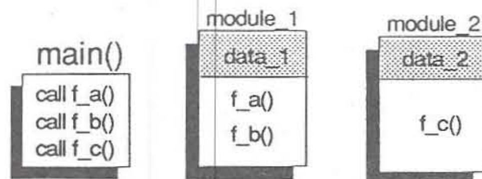


Figure 1: Programmazione modulare.

I sistemi di tempo reale sviluppati per mezzo di tecniche tradizionali presentano usualmente forti criticità. Tali sistemi risultano mal strutturati poiché i vincoli temporali in molti casi costringono lo sviluppatore a compromessi per i quali ogni porzione di codice è nella maggior parte dei casi strettamente collegata ad altre (a causa dei vincoli temporali) che sono concettualmente disgiunte. Pertanto, il problema principale dei metodi tradizionali per la specifica dei sistemi di tempo reale è che questi non sono abbastanza formali da permettere una modellazione strutturata anche in presenza di vincoli temporali. Questo va a scapito del riuso; inteso sia come riuso delle specifiche che come predisposizione della specifica in modo che questa possa essere facilmente riutilizzata in futuro.

3 Cenni Storici

Al fine di rendere facilmente assimilabili i concetti che verranno discussi nei prossimi paragrafi, in questa sessione viene brevemente riportata l'evoluzione dei concetti che sono alla base del paradigma di programmazione orientato agli oggetti.

3.1 Programmazione Modulare

In base alla programmazione modulare, per mezzo di un processo di astrazione, il problema in esame viene decomposto per identificare problemi più piccoli che possono essere più facilmente risolti, e quindi composti per portare alla soluzione del problema completo. Il modulo incapsula al suo interno dati ai quali si dovrebbe accedere solo tramite procedure o servizi del modulo (si veda Fig.1). La strutturazione in moduli del problema conferisce una maggiore robustezza, estendibilità, riusabilità, manutenibilità, testabilità [52], [35], [78]. Questo tipo di programmazione, viene detta anche *Object-Based*, ed è spesso confusa con l'OOP o con il concetto di file.

In generale, i concetti della programmazione modulare vengono spesso mal interpretati. Spesso si ha una degenerazione per la quale si cercano di individuare: porzioni di codice ripetute per creare delle procedure che possono essere riutilizzate in più situazioni. L'uso di queste procedure viene spesso generalizzato per parametrizzazione (e.g., aggiungendo opportuni parametri). Tali procedure potranno essere utilizzarle su dati diversi come se fossero uguali. Si finisce pertanto per definire relazioni N:1, N tipi di dato manipolabili da una stessa procedura. Questo porta alla nascita di problemi di congruenza fra tipi, poiché non è sempre possibile separare in modo chiaro i vari comportamenti che la procedura deve presentare nelle varie condizioni. Anche il posizionamento delle procedure nei moduli ha la sua importanza, spesso viene fatto in base a considerazioni che sono solo di comodo e non in base a considerazioni concettuali. Questo si paga con la perdita di portabilità e riutilizzabilità del codice.

Pertanto, se la programmazione modulare non è assistita da linguaggi specifici, i moduli perdono la propria autonomia degenerando dalla fase di progetto a quella di realizzazione, passando da una

programmazione modulare ad un modello a *spaghetti*. Maggiori garanzie si hanno utilizzando linguaggi che offrono la possibilità di definire nuovi tipi di dato in modo completo, definendo: le possibili operazioni sui tipi, le conversioni fra tipi (i.e., *casting*) implicite o esplicite, e garantendo un controllo stretto della congruenza fra tipi (i.e., *strong typechecking*).

3.2 Abstract Data Type

Secondo il concetto di Abstract Data Type, ADT, un tipo di dato è composto da una struttura dati più un certo numero di operazioni per manipolarli. I dati sono incapsulati realizzando il meccanismo di *data hiding* (i.e., *information hiding, encapsulation*) [40], [29], [39], per il quale le operazioni sono la sola interfaccia per manipolare il contenuto informativo dall'esterno. I concetti di ADT e quello di modulo sono 'simili' ma negli ADT i meccanismi per garantire il *data hiding* sono definiti in modo formale, impedendo al programmatore interpretazioni personali come invece può accadere nelle programmazione modulare.

Dal punto di vista formale, prima di utilizzare un nuovo tipo di dato questo deve essere definito e realizzato. La definizione di un ADT consiste nella dichiarazione: (i) del dominio delle operazioni del dato e (ii) dell'interfaccia per l'uso di tali operazioni. Tale definizione deve essere indipendente dalla realizzazione interna del dato e dal tipo di linguaggio, per esempio, la definizione del tipo `Stack` consiste in:

```
Stack
    Elemento := Pop(Stack);
    Push(Stack,Elemento);
    Flag := Is_Empty(Stack);
    Stack := New();
```

mentre la realizzazione si sviluppan nella (i) definizione della struttura dati del tipo (e.g., lo `Stack` come un vettore di elementi o come una lista) a partire da tipi già definiti, e nella (ii) codifica delle procedure di cui è stata definita l'interfaccia. Le procedure descrivono il funzionamento interno del'ADT (e.g., FIFO, LIFO, etc.) e possono essere realizzate tramite linguaggi algebrici, logici, etc.

Uno dei vantaggi dell'ADT consiste nel fatto che si possono utilizzare più istanze dello stesso tipo senza doverne tenere conto nella sua struttura, come invece spesso accade nella programmazione modulare. Secondo l'approccio ADT si possono sovrascrivere le operazioni, cioè ADT indipendenti possono avere operatori, funzioni aventi lo stesso nome (si possono pertanto definire operazioni polimorfiche, cioè funzioni che lavorano su tipi diversi). Attraverso la sintassi il compilatore sarà in grado di riconoscere a quale operazione si intende riferirsi – e.g., $< Scalare > \cdot < Vettore >$, $< Vettore > \cdot < Matrice >$ l'operazione di prodotto scalare assume significati semantici diversi in funzione del tipo dei suoi operandi.

Nei metodi basati su ADT il programmatore è costretto a rispettare i canoni della programmazione modulare usufruendo così dei relativi vantaggi: *data hiding*, un controllo stretto dei tipi, e il polimorfismo, etc.

4 Object-Oriented Paradigm per Sistemi di Tempo Reale

Per Object-Oriented Paradigm (OOP) si intende l'insieme dei meccanismi che definiscono la metodologia di modellazione ad oggetti [8], [19]. Questo paradigma è la naturale evoluzione dei concetti

della programmazione modulare e basata su ADT. In effetti, l'OOP non è solo un paradigma di programmazione, ma un modo di modellare sistemi e di affrontare il problema della modellazione, molto diverso dall'approccio strutturato tipico dei DFD (Data Flow Diagram) [23] o delle sue estenzioni per i sistemi di tempo reale che ben si sposano con i concetti dell'ADT [31], [71]. Secondo l'OOP l'attenzione si sposta, come per l'ADT, dalle trasformazioni dati (cioè dall'aspetto funzionale) a quello strutturale e comportamentale dei tipi di dati [20], [21], [45], [46].

In modo molto simile agli ADT, secondo l'OOP i "tipi" di dato vengono ottenuti in due fasi distinte: la *definizione* e la *realizzazione*. La *classe* è la struttura formale attraverso la quale si *definiscono* i nuovi "tipi di dati" del paradigma. Questo concetto è simile al concetto di ADT. "tipi" poiché in effetti il concetto di tipo e di classe non sono coincidenti [21], [48]. La *definizione* della classe consiste nel dichiarare la struttura dati (i.e., gli attributi della classe) e i domini delle operazioni che possono essere effettuate sul tipo sotto definizione (i.e., le operazioni, i metodi della classe). La *realizzazione* consiste nel progetto e nella codifica dei metodi, questi sono l'unica interfaccia che le altre entità software del sistema possono vedere realizzando il meccanismo di *data hiding*. Per esempio:

```
Class Stack
begin
Attribute:
    Tipoele *elemento;
    Integer indice;
    Integer dimensione;
Method:
    Stack Create(Integer); // allocazione (Costruttore), istanziazione
    Boolean Push(Tipoele);
    Tipoele Pop();
    Change(Tipoele *, Integer, Integer);
    Boolean Is_Empty();
    Delete();              // deallocazione (Distruttore)
    .......
end;
```

I metodi di una classe possono essere classificati in *costruttori* (metodi ove è possibile definire il meccanismo di creazione delle nuove istanze – i.e., allocazione), *distruttori* (metodi ove è possibile definire il meccanismo di distruzione delle istanze – i.e., deallocazione), *selettori* (metodi che servono per leggere lo stato di un oggetto – i.e., leggere il valore degli attributi), *operatori* (unari – e.g., NOT, cambia di segno, ABS, etc.; binari – e.g., +, -, *, /, AND, OR, MOD, ==, >=, etc.), etc.

La realizzazione vera e propria della classe consiste nella codifica dei metodi il cui dominio, interfaccia, è stata definita del corpo della classe. Nella codifica dei metodi viene definito il comportamento della classe, per esempio:

```
Stack Stack::Create(Integer dim)
begin
    elemento := alloca( sizeof(Tipoele), dim);
    indice := 0;
    dimensione := dim;
end;
```

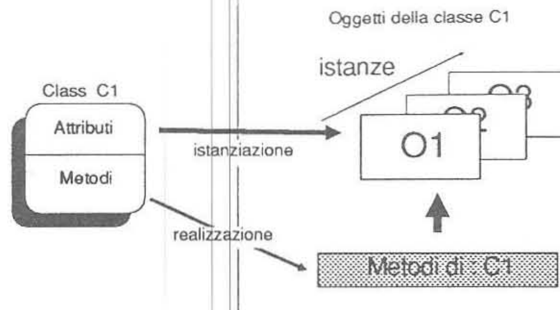


Figure 2: Relazioni fra definizione e realizzazione secondo l'OOP. La classe C1 ed alcune sue istanze.

```
Boolean Stack::Is_Empty()
begin
    if ( elemento==NULL ) then return(TRUE);
    else return(FALSE);
end;

Boolean Stack::Push(Tipoele ele)
begin
    if (indice+1<<=dimensione) then begin
       indice := indice+1;
       elemento[indice] := ele;
       return(TRUE);
    end;
    else return(FALSE);
end;
```

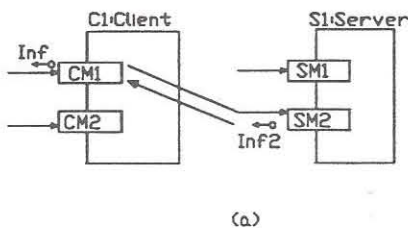
Secondo il paradigma orientato agli oggetti la chiamata ad un metodo per la sua esecuzione viene vista come l'invio di un messaggio ad un oggetto affinché questo esegua l'azione corrispondente – e.g., con `S1.Push(E)` viene inviato allo `Stack S1` il messaggio "inserisci E", questo potrà suscitare o meno una reazione da parte dello `Stack S1`. La realizzazione di questo paradigma di *message passing* comporta la definizione del tipo di protocollo per la comunicazione fra oggetti. In realtà in linguaggi dove non si ha la concorrenza fra istanze non è corretto parlare di messaggi ma solo di chiamate di metodi che in questo contesto sono procedure.

Attraverso il processo di *istanziazione* si creano gli oggetti (i.e., le istanze) di una data classe stabilendo in questo modo un legame formale del tipo IS-INSTANCE-OF fra oggetto e classe. Ogni oggetto ha una struttura dati definita dagli attributi della classe, mentre su di esso possono essere eseguiti i metodi, questi al momento dell'esecuzione sono condivisi da tutte le istanze della classe (si veda Fig.2). Il processo di istanziazione di un oggetto può avvenire a partire dalla classe o da un altro oggetto, inoltre, l'istanziazione può essere sia statica (al momento del caricamento del codice eseguibile) che dinamica (al tempo di esecuzione). Il processo di istanziazione dinamica coincide con l'allocazione dinamica di memoria, pertanto si può avere un numero di oggetti variabile, e di conseguenza si hanno problemi di *garbage collection* per recuperare e riorganizzazione delle aree libere di memoria che vengono a crearsi al momento della distruzione di oggetti tramite de-allocazione.

Si noti che secondo l'OOP, il concetto di classe non è estensionale come nel linguaggio corrente. Nel senso che, manipolare la classe non significa manipolare tutte le entità che ne fanno parte ma

```
Class Client
    Methods:
        Inf_Type CM1();
        CM2();
end;
```

```
Class Client
    Provided_services:
        CM1 : CM1_Type;
        CM2 : CM2_Type;
        Inf2 : Inf2_Type;
    Required_services:
        ReqInf2 : ReqInf2_Type;
        Inf : Inf_Type;
        ReqX : ReqX_Type;
end;
```



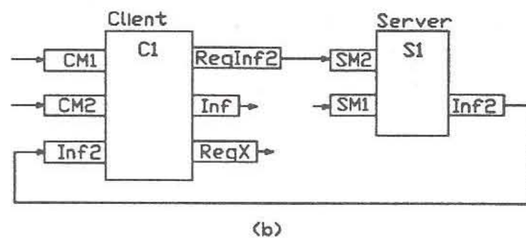


Figure 3: Confronto fra i modelli di interfaccia di tipo *method-based* (a) ed *IO-based* (b) nel passaggio di messaggi fra oggetti. Nel caso (a) il metodo `CM1()` nella sua codifica conterrà una chiamata al metodo `SM2()` della classe `Server`.

solo la descrizione della categoria che la classe descrive, la classe non conosce quante istanze della classe stessa (i.e., oggetti) sono state create. Questo attribuisce alla classe il compito di essere una specie di stampo per la struttura dati degli oggetti, e non un supporto per indicizzare gli oggetti creati.

Si deve notare che la descrizione dell'interfaccia della classe basata su metodi non è adatta a rappresentare tutte le relazioni che la classe ha con altri oggetti e classi. Poiché, tale descrizione, non presenta in modo esplicito quali sono i servizi che la classe richiede all'esterno per esempio tramite la chiamata ad un metodo di un'altra classe. Tali richieste sono nascoste nella realizzazione del metodo [76], [70]. Osservando l'interfaccia della classe si possono vedere solo i servizi che la classe può offrire (*provided service*) e non quelli che la classe richiede ad altri oggetti di altre classi (*required service*). Questa è una grossa limitazione per il riuso della classe stessa, che non può essere semplicemente estratta dalla libreria poiché la classe stessa dipende dalle classi alle quali richiede servizi. Una soluzione a questo problema è l'utilizzo di un'interfaccia per la classe di tipo *IO-based* come è stato proposto in alcuni metodi e linguaggi Object-Oriented ed Object-Based [5], [13], [18]. In Fig.3 è riportato il confronto fra l'approccio basato su metodi e quello basato su IO. L'approccio IO-based è anche molto efficiente per la specifica dei sistemi di tempo reale, poiché utilizzando questo tipo di interfaccia possono essere più facilmente identificate le relazioni fra i vari processi del sistema e quindi i vincoli temporali.

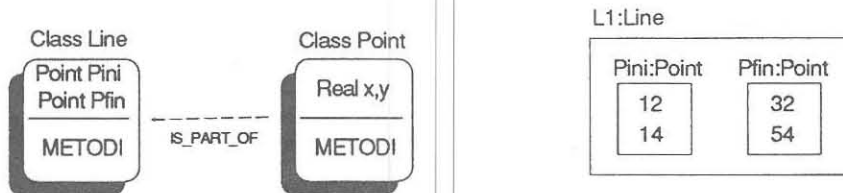


Figure 4: (a) Relazione fra le Classi `Line` e `Point`, (b) istanza della classe `Line` con le sue sottoparti.

4.1 Relazioni di aggregazione e associazione

Quando vengono dichiarati gli attributi della classe (che sono a loro volta istanze di classi) si definiscono in modo implicito i legami di aggregazione e di associazione fra classi – per esempio uno `Stack` può essere realizzato come un'aggregazione di elementi. Questi legami si concretizzano solo quando si istanziano gli oggetti. La relazione di aggregazione definisce un legame di IS-PART-OF (è parte di) fra classi, tale legame viene usualmente rappresentato con una linea tratteggiata fra classi. Per esempio dopo la fase di istanziazione dell'oggetto `L1` della classe `Line` si presenta la situazione di Fig.4 in cui `Pini` e `Pfin` sono attributi della classe `Line`:

```
Class  Line
begin
Attribute:
    Point Pini;
    Point Pfin;
Method:
    Line  Create(Point, Point);
    Set(Point, Point);
    Point Get_Pini();
    Point Get_Pfin();
end;
```

e quindi a livello di oggetti una linea è l'aggregazione di due punti che sono le sue sottoparti. Mentre a livello concettuale fra la classe `Line` e `Point` esiste un legame di aggregazione (IS-PART-OF).

Quando la relazione di aggregazione fra oggetti viene realizzata tramite un legame di tipo IS-REFERRED-BY significa che la sottoparte è referenziata attraverso un puntatore, questo permette l'allocazione e/o il cambio dinamico di una sottoparte. Si possono pertanto definire per mezzo di questo meccanismo condizioni di condivisione di uno stesso oggetto (*object sharing*, *data sharing*) (si veda Fig.5). Questo tipo di legame può creare problemi al momento della cancellazione di un referente che a sua volta può implicare la cancellazioen dell'oggetto condiviso e quando si è in un sistema concorrente quando più processi accedono e modificano oggetti condivisi.
Per mezzo della relazione di associazione (legami di tipo IS-REFERRED-BY) fra oggetti si possono definire legami 1:N fra oggetti. Tali relazioni sono gestite da particolari classi, dette liste o collezioni. Queste possono gestire oggetti dello stesso tipo (associazioni omogenee) di tipo diverso (associazioni eterogenee). In questo secondo caso si è in presenza di una classe polimorfica che, come verrà

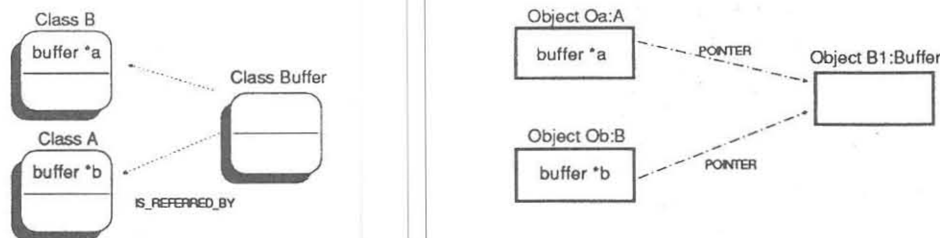


Figure 5: Uso di relazioni di tipo IS-REFERRED-BY per la condivisione di oggetti. (a) Relazioni fra classi, (b) relazioni fra oggetti, gli oggetti `Oa` e `Ob` referenziano il **`Buffer B1`**.

mostrato in seguito, può essere realizzata in modo efficiente solo attraverso particolari meccanismi di ereditarietà.

Nella specifica di sistemi di tempo reale si deve fare molta attenzione alla definizione di legami di tipo IS-REFERRED-BY, specialmente quanto questi sono definiti per poter allocare dinamicamente oggetti, realizzare liste di oggetti e condividere di oggetti. L'allocazione dinamica in genere deve essere evitata per conferire al sistema di tempo reale un maggiore determinismo, sui tempi di esecuzione e sull'uso della memoria. Questi problemi sono presenti anche negli approcci di tipo Object-Based.

4.2 Ereditarietà

L'*ereditarietà* è il meccanismo più importante dell'OOP; questo meccanismo ed i suoi annessi sono i fattori che maggiormente differenziano l'OOP dall'ADT, questo fatto viene usualmente schematizzato con l'ugualianza:

$$OOP = ADT + EREDITARIETA'$$

Utilizzando i meccanismi di ereditarietà si definiscono delle relazioni concettuali fra classi. Per esempio in Fig.6a si ha che `Vehicle` è la superclasse della classe `Car` e quindi la classe `Car` è la sottoclasse della classe `Vehicle`. La definizione di relazioni di ereditarietà permette di definire in modo formale le relazioni di specializzazione/generalizzazione (astrazione) di tipo concettuale fra categorie di entità [62], [65], [66]. Queste relazioni sono usualmente rappresentate attraverso alberi della classi (*class tree*), nei quali compaiono le relazioni di specializzazione (IS-A), e possono comparire anche le relazioni di aggregazione (IS-PART-OF) e di associazione (IS-REFERRED-BY). In Fig.6b si ha che un Duetto Alfa Romeo è una `Spider` ma può essere considerato in modo più generico una `Car` o un `Vehicle` (ma non il viceversa). Il legame IS-PART-OF specifica che in genere i veicoli hanno ruote. In questo caso si dice che una `Spider` è un membro della classe `Vehicle`,

L'ereditarietà consiste nella condivisione monodirezionale fra classi (i) della struttura dati (i.e., realizzazione del meccanismo di *structure sharing*), e (ii) del comportamento (i.e., realizzazione del meccanismo di *behavior sharing*). Quando vi è una relazione di ereditarietà fra classi si ha che la sottoclasse eredita dalla sua superclasse sia gli attributi che i metodi. Pertanto la definizione della sottoclasse è equivalente ad aver definito una classe senza padre con (i) gli attributi propri più gli attributi della sua superclasse in modo ricorsivo fino alla radice, e (ii) i metodi propri più i metodi

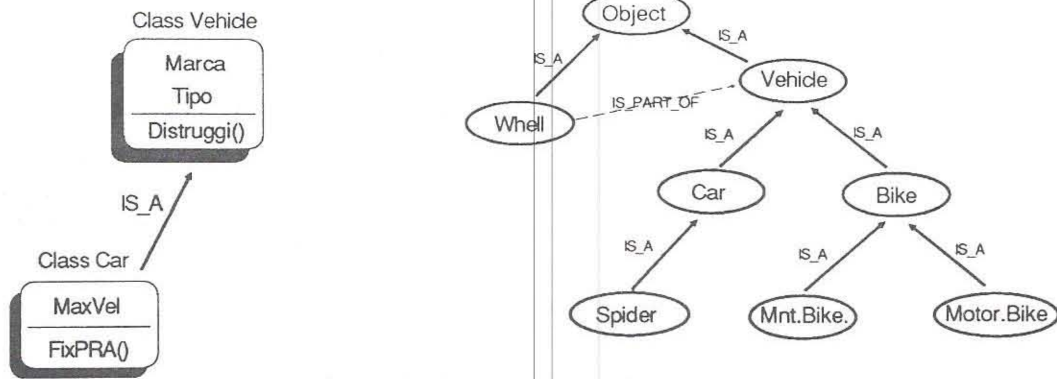


Figure 6: Ereditarietà: (a) relazione si specializzazione fra classi, (b) albero delle classi – i.e., *class tree*.

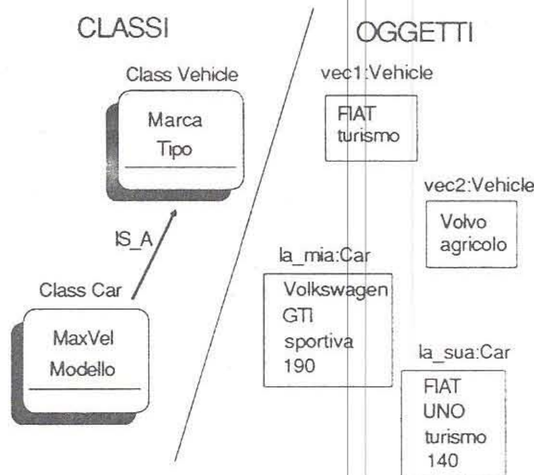


Figure 7: Classi in relazione di specializzazione e relativi oggetti.

della sua superclasse in modo ricorsivo fino alla radice. Si deve notare che l'ereditarietà definisce un legame di specializzazione dal punto di vista delle astrazioni (la sottoclasse è una specializzazione della superclasse), ma risulta essere un'estensione dal punto di vista della struttura della classe, visto che nella sottoclasse si possono aggiungere nuovi attributi e metodi.

Al momento della creazione dell'oggetto si ha la concretizzazione dei legami: IS-PART-OF, IS-REFERRED-BY (associazione con altri oggetti, relazioni 1:N) che erano solamente definiti a livello di classe, mentre le relazioni di ereditarietà definite fra classi non si ripercuotono fra gli oggetti (si veda Fig.7).

Nel processo di specializzazione un attributo di una superclasse può essere sovrascritto da un attributo omonimo definito nella sottoclasse. In questo caso si parla di sovrascrittura (i.e., *overriding*). Se l'ereditarietà è strettamente monotona sono possibili le sovrascritture solo quando la classe stessa dell'attributo subisce una specializzazione. Un meccanismo analogo può essere definito per i metodi e gli operatori (i.e., *overloading*). Anche le operazioni di uso comune (e.g., +, -, ==, etc.) possono essere sovrascritte con altre che vengono messe in esecuzione in base alla

Figure 8: Uso del polimorfismo, ereditarietà e classe astratta.

sintassi dell'espressione. Si può in questo modo creare metodi (polimorfici) che possono essere utilizzati su oggetti di classi diverse. Nell'uso delle sovrascritture e dell'ereditarietà in genere si deve fare molta attenzione, poiché in certi casi l'utilizzo di un'istanza di una sottoclasse come se fosse un'istanza della sua superclasse può causare gravi errori di incongruenza, per esempio quando si hanno sovrascritture e metodi ereditati che lavorano su attributi sovrascritti.

La definizione di una superclasse comune a più classi può essere utile per raccogliere le caratteristiche comuni di una data categoria di oggetti (generalizzare) anche se non si ha intenzione di istanziare oggetti della classe generica. Questo meccanismo porta a definire classi astratte (i.e., *generic, abstract class*). Il concetto di polimorfismo è strettamente legato a quello di ereditarietà ed a quello di classe astratta. Per esempio in Fig.8, la classe `Garage` è polimorfica poiché può manipolare sia gli oggetti della classe astratta `Vehicle` che quelli delle sue sottoclassi. Quindi attraverso un legame IS-REFERRED-BY si possono referenziare oggetti di classi diverse purché organizzate secondo una gerarchia IS-A. Riguardo all'uso dei metodi polimorfici, in alcuni linguaggi si può decidere se utilizzare un meccanismo di risoluzione delle chiamate di tipo statico (eseguito al momento della compilazione, *static binding*) o dinamico (eseguito in fase di esecuzione, *dynamic or late binding*).

L'ereditarietà può essere multipla, nel senso che una sottoclasse può avere più superclassi. In genere, deve essere fatta molta attenzione alla definizione di *eredità multiple* poiché spesso la necessità della definizione di un legame di IS-PART-OF viene confusa con quella di avere un legame IS-A. In presenza di eredità multiple si hanno problemi di conflitti quando sono presenti in più di una superclasse attributi e metodi omonimi. Osservando l'esempio di Fig.9, non si può dire quale metodo sarà eseguito in caso di chiamata del metodo `Copy()` per un'istanza della classe `ModelloGrafico`. Attraverso i meccanismi di ereditarietà ci si può trovare in situazioni di incongruenza quando sono consentite dal punto di vista sintattico delle operazioni su oggetti della sottoclasse solo perché queste sono definite nella superclasse. Sulle istanze della classe `ModelloGrafico` è consentito l'uso del metodo `Move()` che però non agisce sulla parte dell'oggetto ereditata dalla classe `Modello`. Oltre a questi problemi si ha che gli attributi ed i metodi definiti in una superclasse possono essere ridefiniti nelle sottoclassi per mezzo dei meccanismi di overriding e overloading. Per esempio nel caso di Fig.9 se viene fatta la ridefinizione del metodo `Copy()` nella classe `ModelloGrafico` quale sarà il metodo ridefinito? o lo sanno tutti e due? Spesso la risoluzione della chiamata può essere facilmente risolta in base alla sintassi, ma se non è possibile, per poter garantire la risoluzione della chiamata si deve ricorrere a meccanismi di ereditarietà parziale o selettiva. Per i quali è possibile indicare a livello di superclasse quali sono gli attributi e/o i metodi che vengono ereditati, oppure a livello di sottoclasse se possono essere ereditate o meno certe categorie di membri della superclasse come in C++.

Nella specifica di sistemi di tempo reale l'uso dell'ereditarietà deve essere accuratamente dosato.

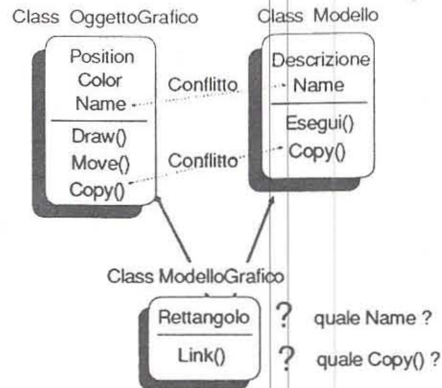


Figure 9: Ereditarietà multipla.

Quasi sempre la definizione di legami di specializzazione implica la presenza di metodi e classi polimorfiche costringendo il sistema ad una risoluzione dinamica delle chiamate (dynamic binding). Come per la creazione dinamica degli oggetti anche la risoluzione dinamica delle chiamate deve essere evitata al fine di rendere più efficiente e predicibile l'esecuzione della specifica del sistema. Questo può essere effettuato non definendo metodi e classi polimorfiche. L'ereditarietà , semplice o multipla, può essere utilizzata solo se la risoluzione delle chiamate ai metodi viene effettuata in fase di compilazione (binding statico). Se tali scelte non possono essere effettuate la risoluzione dinamica delle chiamate deve essere evitata almeno per le classi e porzioni di codice che sono coinvolte in vincoli temporali stretti. In questo modo si può garantire una maggiore predicibilità dei tempi di esecuzione del sistema.

4.3 Type vs Class

A prima vista il concetto di tipo e classe possono sembrare sinonimi, in effetti lo sono solo in casi particolari. Il tipo di una classe è definito in base alla definizione della struttura dati della classe stessa. Pertanto, vi possono essere classi concettualmente distinte che sono identiche come tipo, per esempio la classe punto (avente come attributi le coordinate di tipo `Real`) è dello stesso tipo di un qualsiasi altra classe che ha solo due attributi di tipo `Real`. In un sistema OOP il problema non si dovrebbe porre poiché tutti i controlli sulla congruenza dei tipi vengono eseguiti sulle classi, pertanto si dovrebbe parlare di *classchecking*. Si deve notare che se l'ereditarietà è monotona una sottoclasse è anche un sottotipo [67], [73]. Fattori che provocano la perdita dell'equivalenza fra i concetti di subclassing (specializzazione) e subtyping (specializzazione monotona) sono la multipla ereditarietà e le forme di ereditarietà parziale. Nella specifica dei sistemi di tempo reale è consigliato l'uso dell'ereditarietà monotona per non incorrere in incongruenze concettuali e comportamentali che possono rendere inutili la verifiche di congruenza.

4.4 Aspetti Avanzati dell'OOP

La struttura di un sistema orientato agli oggetti è intrinsecamente parallela poiché dal punto di vista concettuale gli oggetti sono entità ben definite che interagiscono tramite messaggi. Nonostante questo concetto sia proprio dell'OOP la concorrenza fra oggetti non è direttamente supportata a livello di paradigma. In particolare, non sono definiti i meccanismi con i quali devono essere gestite le comunicazioni fra oggetti concorrenti, le regole per la condivisione di oggetti, etc. La presenza

o meno di un supporto per la concorrenza dipende dal linguaggio e dall'ambiente integrato che si utilizza per la specifica [1], [77], [33], [14]. Nella maggior parte dei casi il supporto per la concorrenza in sistemi orientati agli oggetti sviluppati con linguaggi come il C++, Objective C, etc. deve essere realizzato da chi sviluppa il sistema [22], mentre molti altri approcci si sono basati su linguaggi già dotati di concorrenza.

Nelle applicazioni reali vi sono pochi oggetti che tengono il controllo e altri che eseguono solo dei comandi. Pertanto, in sistemi *object-oriented concorrenti* vengono identificati gli oggetti attivi e passivi [3], [33], [77]. Un oggetto è *attivo* quando detiene una certa autonomia ed il controllo di altre attività, eseguendo azioni senza che queste siano dovute esplicitamente a messaggi di altri oggetti. Un oggetto è *passivo* quando esegue azioni solo su richiesta di altri oggetti. Sia gli oggetti attivi che passivi possono essere realizzati con *thread* di esecuzione distinti (dove i passivi seguono il paradigma *event-driven*). La suddivisione del sistema in oggetti attivi e passivi aumenta il parallelismo del sistema ma lascia al programmatore la risoluzione dei problemi di sincronizzazione. La necessità di specificare quali sono gli oggetti attivi e passivi del sistema è solo un artefatto poiché dal punto di vista concettuale fra questi tipi di oggetti non vi è nessuna differenza. Pertanto dovrebbe essere la specifica stessa del comportamento dell'oggetto ed il contesto a determinarne la natura. Pertanto, l'approccio dovrebbe identificare in modo automatico quali sono gli oggetti attivi. Quando si parla di un sistema orientato agli oggetti concorrente si può parlare in modo corretto della realizzazione del paradigma *message passing*. Dal punto di vista formale la chiamata di un metodo di una classe comporta l'invio di un messaggio ad un oggetto. Il protocollo di comunicazione fra oggetti può prevedere o meno l'attesa della risposta da parte del mittente in questo caso si può parlare di sincronizzazione. In caso di oggetti concorrenti che comunicano in modo asincrono il meccanismo di smaltimento dei messaggi deve essere realizzato per mezzo di code [69].

La concorrenza in un sistema ad oggetti può essere realizzata in vari modi, a livello di oggetto (*ogni oggetto un thread* di esecuzione) oppure a livello di metodo (*ogni metodo un thread* di esecuzione) [22]. Nel primo caso vi è la necessità di avere un gestore dei messaggi, dei comandi, che devono essere svolti dall'oggetto. In questo caso la comunicazione asincrona può essere semplicemente realizzata attraverso un'unica coda di messaggi per raccogliere le richieste per tutta la classe. Nel secondo caso, ad ogni metodo viene associato un *thread* di esecuzione che può accedere in modo concorrente agli attributi dell'oggetto. Per non creare conflitti gli accessi devono essere regolati attraverso opportuni semafori di sistema. Si deve notare che un sistema ad oggetti concorrenti può essere visto come un sistema distribuito, e da questo, nel caso che si desideri passare ad una reale distribuzione degli oggetti su processori diversi, è necessario modificare solo i meccanismi di comunicazione.

Si dicono sistemi completamente ad oggetti (*fully object-oriented*, FOO) quelli dove tutti gli elementi software sono visti come istanze di una classe. In questi sistemi si ha (i) un'unica classe radice (chiamata usualmente `Object` o `Root`), che è istanza di sè stessa (questo fatto non aiuta il riuso se non fra sistemi con radici simili); (ii) la presenza della metaclasse, classe le cui istanze sono le classi del sistema (questa ha come metodi: `add_metod()`, `add_attribute()`....); (iii) la presenza delle classi degli attributi (tipi base) e dei metodi; (iv) la possibilità di navigare fra classi e fra classi e oggetti (spesso la classe ha un significato anche estensionale); (v) la persistenza dei legami e della situazione (istanze) nel tempo; (vi) binding dinamico; (vii) *subclassing* dinamico (*dynamic inheritance*), meccanismo per mezzo del quale un oggetto può vedersi cambiare a run-time il valore di un suo attributo ereditato poiché viene cambiata la definizione di tale attributo nella classe

dal quale è stato ereditato[1] [48]. A causa della presenza di queste caratteristiche gli ambienti di tipo FOO sono molto efficienti per la realizzazione di prototipi e quindi per seguire un approccio *prototyping*. Secondo questo tipo di approccio le parti principali del sistema vengono realizzate subito (anche se come scatole vuote) e messe ad interagire, realizzando in questo modo solo gli aspetti comportamentali globali del sistema. Mentre quelli trasformazionali vengono sviluppati in una seconda fase, nella fase iniziale è sufficiente sapere che il dato verrà trasformato a fronte di certi comandi, etc. Questo approccio permette di identificare quali sono le criticità del sistema nel suo complesso ed è molto utile per la specifica di sistemi di tempo reale se opportunamente supportato da verifiche formali. Si deve notare però che la maggior parte dei metodi che supportano in modo estensivo l'approccio prototyping sono ambienti completamente dinamici ed interpretati, questo conferisce una scarsa predicibilità del comportamento rispetto ai vincoli temporali, pertanto li rende inutilizzabili per la specifica di sistemi di tempo reale.

5 Tool Object-Oriented per Sistemi di Tempo Reale

In questa sessione sono brevemente presentati i principali approcci orientati agli oggetti per la specifica di sistemi di tempo reale; questa introduzione è seguita da una discussione sulle metodologie e sul ciclo di vita dei sistemi di sistemi di tempo reale orientati agli oggetti.

Negli ultimi 20 anni molti degli approcci preesistenti adatti alla specifica del comportamento dei sistemi (e.g., Z, VDM (Vienna Development Method [37]), SDL (Specification and Description Language [54]), etc.) sono stati integrati utilizzando altre tecniche per migliorare le loro capacità nel modellare anche gli aspetti funzionale e strutturale e per dotarli di un supporto per il riuso. In molti casi, queste caratteristiche sono state introdotte adottando il paradigma orientato agli oggetti (e.g., Z++, VDM++, OSDL, Objectcharts, TRIO+, etc.). L'adozione dell'OOP ha aggiunto la capacità di strutturare il sistema (per mezzo del concetto di classe), e quella di riusare le specifiche (per mezzo dei concetti di ereditarietà, polimorfismo, e istanziazione). Questo andamento è presente sia per i metodi descrittivi che per quelli operazionali. Fra questi vi sono sia i metodi basati su macchine a stati o reti di Petri che quelli basati su *altre notazioni*.

Fra i metodi *descrittivi* ad oggetti più noti si possono identificare quelli derivati da Z – e.g., OOZE [2], Z++ [38], Object-Z [16] e da VDM – e.g., VDM++ [26]. La maggior parte di questi supporta il data hiding, l'ereditarietà, e il polimorfismo. Fra questi Object-Z integra anche i concetti della logica temporale. In questo approccio lo stato è una sorta di storia di eventi che descrivono il comportamento dell'oggetto, pertanto l'approccio è più operazionale che nelle prime versioni di Z. Anche il VDM++ permette la definizione di vincoli temporali. Sfortunatamente, questi linguaggi non sono direttamente eseguibili pertanto rimangono ancora confinati negli istituti di ricerca e sono oggetto di studio nelle maggiori industrie europee per valutare la loro reale applicabilità.

Fra i metodi *operazionali* ad oggetti sono da privilegiare quelli derivati dalle macchine a stati finiti o da reti di Petri poiché sono basati su fondamenti matematici ormai consolidati. La maggior parte di questi modelli sono IO-based e pertanto sono adatti sia per modellare gli aspetti comportamentali che strutturali del sistema.

[1] questo meccanismo è tipico degli ambienti interpretati dove la struttura delle classi può essere mutata quanto il sistema è in esecuzione.

Fra questi vi è un'interessante estensione di SDL [54] (i.e., SDL 1992) detta OSDL [10]. In questo caso l'SDL con il quale il sistema viene modellato per mezzo di processi e macchine a stati estese è stato integrato con l'OOP aggiungendo i meccanismi di ereditarietà, polimorfismo sia a livello di blocco che di macchina a stati; favorendo il riuso delle specifiche e rendendo più formale l'approccio. Nonostante questo cambiamento radicale, non sono state aggiunte verifiche di proprietà e meccanismi per la definizione accurata dei vincoli temporali e pertanto rimane un approccio completamente operazionale con il quale si giunge alla validazione finale solo per mezzo di simulazioni reiterate. Pertanto non è adatto per la specifica di sistemi di tempo reale stretto.

Le reti PROT sono un approoccio operazionale basato su di un'estensione del modello SPN (reti di Petri stocastiche). Per mezzo di tale modello si possono identificare condizioni critiche per mezzo di meccanismi matematici [11]. Queste reti possono essere facilmente tradotte in ADA. Per queste reti è possibile effettuare la simulazione per validare la specifica e sono supportate da un CASE tool chiamato ARTIFEX [4], e dalla metodologia PROTOB [5]. Di questo approccio viene discusso in modo esaustivo in un altro intervento di questo stesso convegno.

Più recentemente sono state presentate delle estensioni ad oggetti del modello Statecharts [30], al fine di migliorare le sue capacità nel modellare gli aspetti funzionali e la descrizione dei vincoli temporali – e.g., Objectcharts [18], Modecharts [36], e ROOMcharts [58] (con un CASE tool chiamato ObjecTime [51]).

Molti dei metodi operazionali basati su altre notazioni (originariamente creati come supporti per metodologie, e.g., DFD (Data Flow Diagram [23]), JSD (Jackson System Development [35]), etc.) hanno seguito un percorso diverso. Questi sono stati integrati con supporti di basso livello per compensare la loro mancanza di formalismo e per coprire tutte le fasi del ciclo di vita della applicazione. Attualmente esistono metodi basati su DFD e macchine a stati (e.g., RTDFD [31]), JSD e macchine a stati (e.g., Entity-Life [57], [56]), DFT e reti di Petri (e.g., IPTES [53]), Entity-Relationships estese e VDM (e.g., ATMOSPHERE [24]), object-oriented e macchine a stati (e.g., Booch [9]), etc. Per molti di questi la verifica dei vincoli temporali e la validazione finale sono ancora un grosso problema. Per esempio quando la definizione dei vincoli temporali è permessa nelle prime fasi di analisi, la loro consistenza non è verificata nel proseguo della specifica fino alla definizine dei dettagli e alla generazione del codice. In altri casi la definizione del vincoli è permessa solo a livello di codice, quando, se questi non possono essere soddisfatti, è troppo tardi e costoso ristrutturare completamente il sistema al fine di garantire il rispetto di tali vincoli. In ogni caso, a dispetto di questi problemi sono stati realizzati svariati CASE tool.

I metodi *duali* sono stati proposti come un compromesso fra i descrittivi e quelli operazionali nell'intento di realizzare metodi in grado di garantire l'esecuzione della specifica come gli operazionali e la validazione di questa si attraverso meccanismi di simulazione che tramite prove di proprietà . Questi linguaggi e tool sono l'argomento di punta per quanto riguarda la ricerca nel settore della specifica dei sistemi di tempo reale, pertanto non sono presenti sul mercato prodotti commerciali che utilizzano tali modelli. Fra questi, i modelli IO-based: TRIO+ (TRIO object-oriented) [41], [42] e TROL [14], [13].

TRIO+ è un linguaggio logico per la specifica di sistemi modulari [41], [42] è nato come l'integrazione di TRIO e dell'OOP. TRIO+ è basato su una logica del primo ordine con costrutti temporali e gestione del tempo di tipo implicito. Poiché TRIO+ è basato sulla programmazione logica il concetto di stato delll'oggetto coincide con quello di storia del motore inferenziale con il

quale viene eseguita la specifica. TRIO+ è un modello eseguibile e supporta anche l'esecuzione di specifiche parziali.

TROL (Tempo Reale Object-oriented Language) è un linguaggio orientato agli oggetti derivato da un modello esteso di macchine a stati. TROL integra i concetti delle macchine a stati estese con costrutti logici del primo ordine per garantire la verifica di proprietà della specifica. Particolari meccanismi permettono di verificare la consistenza fra i vincoli temporali, i predicati logici e le macchine a stati [14]. TROL fornisce anche supporto per la verifica formale della composizione strutturale di oggetti considerando gli aspetti comportamentali e i loro vincoli temporali, pertanto permette la verifica incrementate della specifica e l'esecuzione di specifiche parziali [13], [15]. Di questo approccio viene parlato in modo più esaustivo in un altro intervento di questo stesso convengo.

5.1 Metodologie Orientate agli Oggetti

Negli ultimi anni l'OOP è stato adottato da molti sviluppatori per supportare il processo di produzione software [17], [9], [51], [18]. Questo è dovuto al fatto che lavorare con classi ed oggetti è un modo naturale di partizionare un problema. I maccanismi dell'OOP rendono facile la decomposizione del sistema in classi ed oggetti, identificando facilmente quali sono le attività che devono essere fatte in parallelo, e che tipo di messaggi si scambiano [19], [25], [6]. Modelli più raffinati hanno diviso gli oggetti attivi e passivi o server/client [1], [27], [9]. Su tale scia, sono stati proposti molti approcci orientati agli oggetti per la specifica di sistemi di tempo reale [72], [55], [17], [43], [61], [34], [6], [18], [51], [49], [32]. Molti di questi approcci sono reinterpretazioni ed estensioni di tecniche tradizionali basate sull'analisi strutturata – e.g., OMT [55], Coad and Yourdon [17], Martin and Odell [43], altri sono approcci completamente ad oggetti Booch [9], and Wirsf-Brock et al. [75]. In molti degli approcci menzionati il sistema è decomposto in oggetti al fine di specificare la struttura del sistema. Le relazioni fra oggetti vengono definite per mezzo di versioni estese dei digrammi *Entity Relationship* [17], [55] o per mezzo di *Object Diagrams* (si veda Fig.10) [9], [55], [32], [75] ove sono riportati i messaggi che vengono scambiati fra oggetti. In molte di queste metodologie il comportamento della classe viene descritto per mezzo di diagrammi a stati estesi, *Extended State Machine* – e.g., Shlaer e Mellor [60], [61] e Booch [9], usano un modello di Mealy; Rumbaugh (OMT) [55] usa una notazione simile agli Statecharts – i.e., ROOMcharts; Coad e Yourdon [17] usano una tabella di eventi. In generale, un oggetto può essere visto come una macchina a stati o una rete di Petri dove lo stato degli attributi rappresenta lo stato della macchina o della rete. In entrambe i casi le transizioni di stato vengono effettuate sulla base di messaggi provenienti dall'esterno oppure a causa di un meccanismo di esecuzione ciclico interno.

Si deve notare che nel caso di approcci completamente ad oggetti la metodologia è molto diversa rispetto a quelle utilizzate per l'approccio strutturato. Le maggiori differenze risiedono nel fatto che secondo la programmazione strutturata l'analista si deve chiedere **come** *deve essere eseguita una certa cosa*, cioè il centro dell'attenzione è sulla trasformazione dati, mentre nell'approccio orientato agli oggetti si deve chiedere **chi**, *o meglio cosa, deve eseguire una certa cosa*. Pertanto, durante l'analisi dei requisiti del sistema si cercano di individuare le classi e gli oggetti reali che compongono il sistema. Le classi vengono identificate sulla base delle tipologie di oggetti reali mentre gli oggetti identificando le loro istanze (e.g., un asse di moto da controllare, l'asse di 35mm posto a 45 gradi, sono la classe ed una sua istanza). Dopo questo fase le varie classi vengono organizzate secondo varie gerarchie: *la strutturale*, relazioni di parte sotto parte e di uso al fine di decomporre il sistema

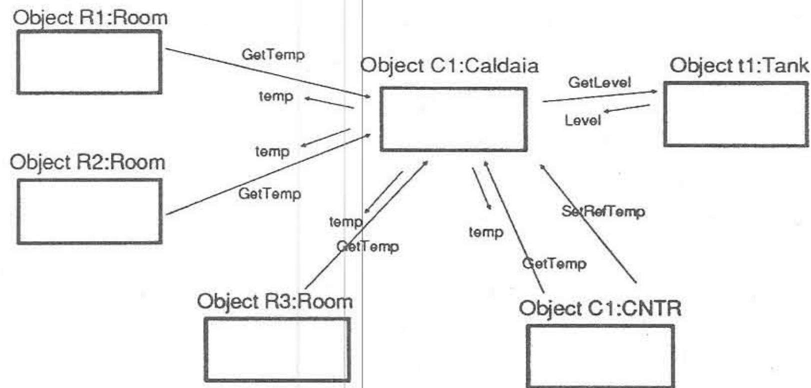


Figure 10: Object Diagram di un controllore distribuito per il riscaldamento domestico.

in classi più piccole (legami IS-PART-OF, IS-REFERRED-BY, etc.); *la concettuale*, relazioni di specializzazione/generalizzazione fra classi al fine di favorire il polimorfismo, etc. Dopo questa fase di analisi si entra nella fase di progetto dove vengono definite nel dettaglio le relazioni e le operazioni, i metodi di ogni classe. Segue la fase di codifica in cui le classi vengono realizzate per mezzo di un linguaggio di programmazione, segue poi il test, etc. Il test come verifica e validazione della specifica può essere presente in varie fasi dello sviluppo in funzione del tipo di approccio che viene utilizzato, sono ovviamente da preferire i metodi per i quali la verifica e validazione può essere eseguita fin dalle fasi di analisi.

In molti tool e linguaggi orientati agli oggetti il processo di verifica formale è solo parziale, e non vi è una vera e propria validazione finale del comportamento del sistema. Questo deriva dal fatto che la maggior parte degli approcci orientati agli oggetti non sono assistiti da un supporto formale matematico anche per quanto riguarda la semantica dell'approaccio. Al fine di mantenere sotto controllo l'evoluzione del sistema, il supporto per la validazione del comportamento del sistema tramite prove di proprietà dovrebbe poter essere utilizzato in qualsiasi momento anche se il sistema è solo parzialmente specificato (favorendo l'approccio prototype), [14]. La maggior parte degli approcci che consentono la validazione attraverso prove di proprietà non consentono di eseguire la specifica, pertanto sono inadatti per la realizzazione di sistemi di tempo reale.

Nella maggior parte dei sistemi orientati agli oggetti si deve notare la mancanza di un supporto per la gestione del comportamento legato ai vincoli temporali, per esempio la definizione di timeout, deadline, etc. Tale supporto dovrebbe coprire sia gli aspetti metodologici che implementativi del paradigma. Per esempio, offrendo la possibilità di definire pre- e post-condizioni sul comportamento della classe dipendenti dal tempo. Pertanto si può affermare che anche la gestione del tempo in un sistema orientato agli oggetti per il tempo reale dovrebbe essere congruente con l'OOP e sufficientemente formale da consentire le verifiche di consistenza e completezza nonché quelle di proprietà con vincoli temporali.

Un compromesso è la possibilità di ottenere la validazione finale del sistema per mezzo di simulazioni. La simulazione può essere basata sull'esecuzione controllata del codice generato oppure sull'interpretazione della specifica a livello semantico. Il primo approccio offre ovviamente maggiori garanzie sul comportamento del sistema finale poiché questo sarà composto dallo stesso codice

che viene utilizzato in simulazione. La simulazione viene usualmente effettuata controllando il comportamento del sistema rispetto ai possibili andamenti dei segnali di ingresso al sistema che possono essere generati manualmente o in modo automatico. I primi danno maggiori garanzie di poter stressare il sistema poiché i secondi possono essere affetti da vizi, e quindi sono da preferire anche se comportano un maggiore sforzo per garantire la copertura di tutti aspetti comportamentali del sistema.

5.2 Ciclo di Vita Orientato agli Oggetti

Nella realtà il ciclo di vita di un sistema orientato agli oggetti non è così semplice come è stato descritto nel precedente paragrafo poiché le fasi di analisi, progetto e test non sono separabili; parti diverse dello stesso sistema si possono trovare nello stesso istante in fasi diverse del ciclo di vita. Per esempio, in un sistema sotto sviluppo possono convivere classi appena analizzate con classi di cui è già stato fatto il progetto e la codifica. Questo accade principalmente poiché l'OOP consente uno sviluppo sia *bottom-up* (partendo dalla classe che modella il sistema) che *top-down* (partendo dalle classi elementari per poi definire quelle che ne fanno uso) rendendo semplice il riuso, e spingendo verso un approccio *prototype*. L'analista esperto è in grado di identificare subito la maggior parte delle classi del sistema finale e le loro relazioni, ma nello stesso tempo è sicuro che durante la fase di progetto saranno individuate alcune nuove classi che porteranno tali parti del sistema nella fase di analisi. Questo processo può essere visto come un'evoluzione del tradizionale ciclo di vita a spirale [7] (si veda Fig.11a). Nella nostra esperienza il ciclo di vita può essere schematizzato come un cono spiralizzato dove l'altezza del cono rapresenta la quantità delle risorse (si veda Fig.11b).

Questa schematizzazione non è del tutto coerente con il ben noto diagramma della productività di Norden [50], dove viene riportato il numero delle risorse rispetto al tempo e lo sforzo è dato dalla superficie sottesa alla curva (si veda Fig.12). Tale andamento parte dall'assunzione che le fasi di analisi, design, etc. sono sviluppate sequenzialmente. In un processo di tipo prototype le varie fasi vengono eseguite per ogni ciclo della spirale anche se ad diverso dettaglio. L'andamento complessivo del numero di persone impiegate nel progetto è in effetti molto simile a quello di Fig.12a, ma nel caso di approccio prototype esso è dovuto dalla somma degli andamenti nei vari cicli. Questi sono a loro volta composti dalle fasi di analisi/reanalisi, design, etc. e sono in parte sovrapposti poiché il passaggio da un ciclo al successivo è graduale (si veda Fig.12c). Comparando tale diagramma con quello di Fig.11 si può notare che lo spazio percorso sull'asse della profondità rappresenta la quantità di risorse della Fig.12c. La velocità con la quale si riesce a raggiungere la base della spirale conica coincide con la pendenza del diagramma di Fig.12c. In Fig.12b è riportato un diagramma che riporta l'andamento del riuso (dovuto al `riuso` delle specifiche astratte, e per mezzo di `specializzazione` e altri metodi orientati agli oggetti in fase di progetto). In base all'esperienza è possibile definire delle curve di percorrenza che permettono di prevedere con un certo margine di errore lo sforzo totale. Naturalemente la previsione deve essere fatta sulle singole curve che modellano i cicli del ciclo di vita a spirale.

A causa di questi fattori la gestione stessa di un progetto orientato agli oggetti deve essere rivista rispetto a quella utilizzata per sistemi tradizionali. In questi, viene usualmente seguita una gestione è di tipo verticale (si veda Fig.13a), dove si ha una supervisione orizzontale per garantire la continuità fra analisi, progetto e codifica con relativo sforzo di produzione di documenti, interpretazione, etc. Un progetto orientato agli oggetti può essere condotto con maggiori profitti e minor sforzo utilizando una gestione di tipo orizzontale dove l'intero progetto viene diviso in sottosistemi o porzioni

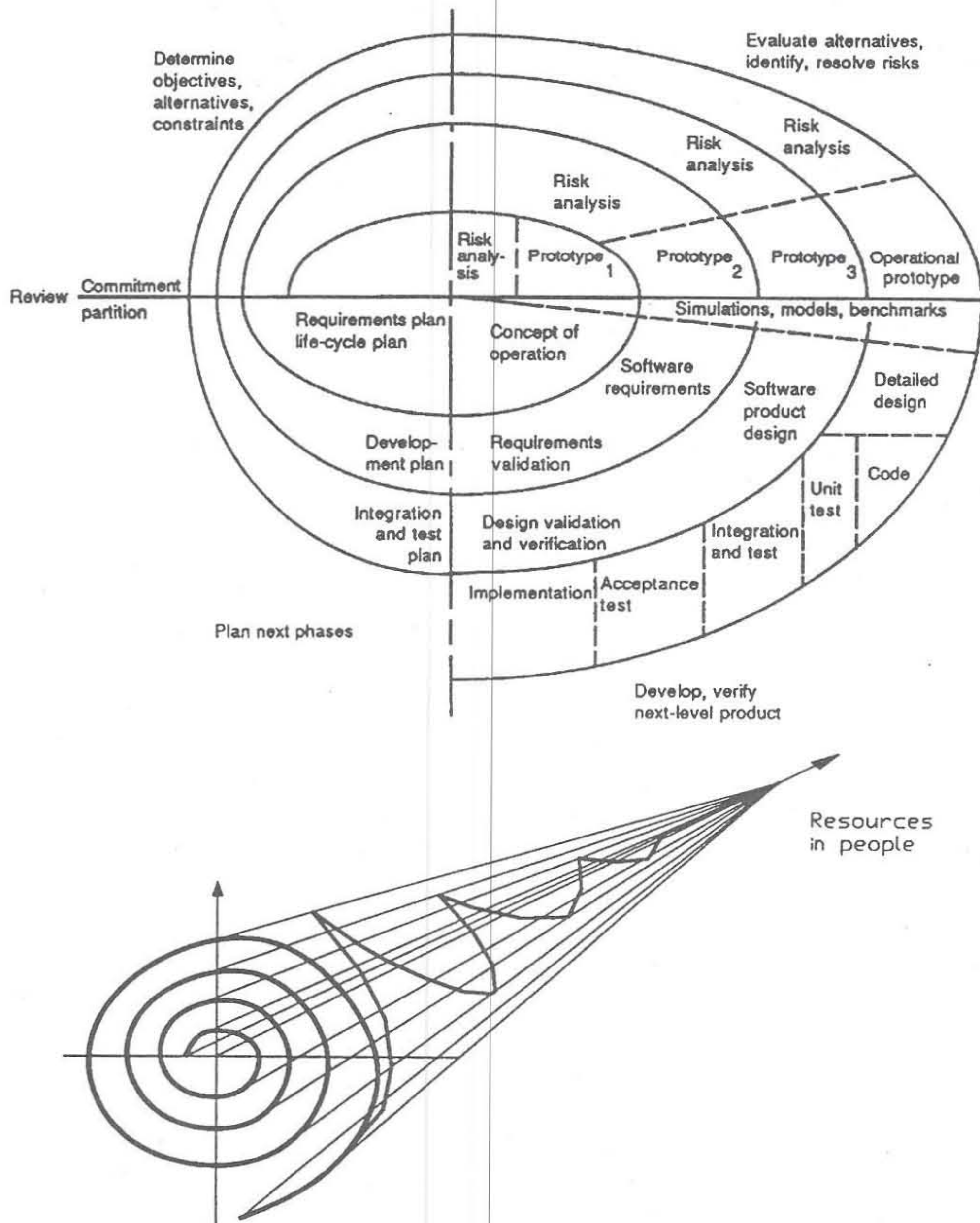
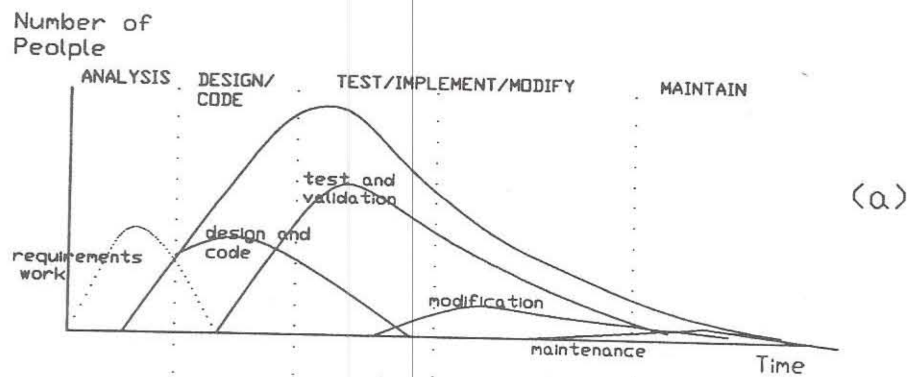
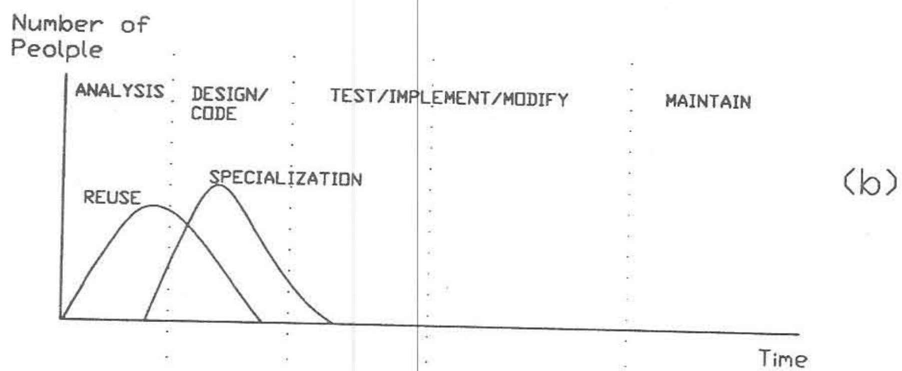


Figure 11: Approccio prototype e ciclo di vita object-oriented.





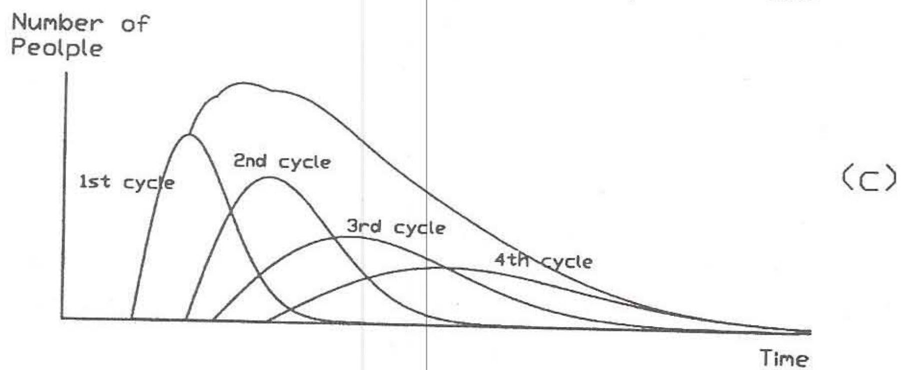


Figure 12: Digrammi di produttività assoluta (a) e dovuta la riuso (b).

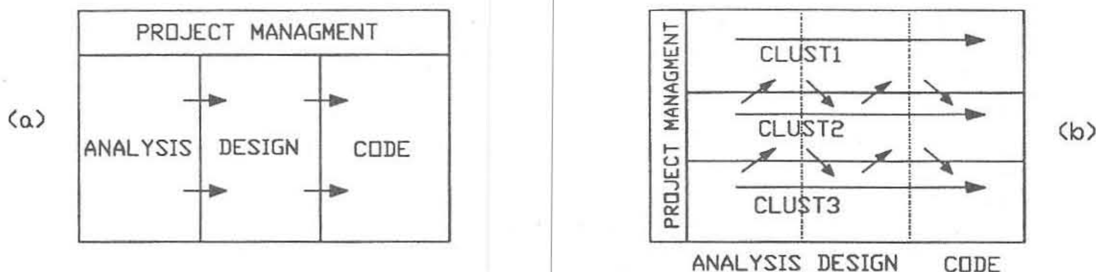


Figure 13: (a) tradizionale gestione verticale, (b) gestione orizzontale orientata agli oggetti.

concettuali di sistema (cluster) che vengono sviluppati dall'analisi alla codifica dalle stesse persone [45], [47] (si veda Fig.13b). In questo tipo di approccio il *Project Manager* deve suddividere in parti il progetto, favorire gli scambi di conoscenze e garantire la continuità della qualità del progetto. In questo caso le persone sono costrette a collaborare e a condividere le conoscenze sul progetto, questo conferma la necessità da parte di chi lavora su progetti ad oggetti di realizzare gruppi di lavoro cooperanti. Il raggiungimento della soddisfazione dei requisiti iniziali del sistema è favorito da un continuo controllo reciproco fra i gruppi di lavoro, dove il *Project Manager* investe il ruolo di moderatore. La figura del Project Manager risulta diversa da quella tradizionale poiché in questo caso egli deve avere una conoscenza del progetto a tutti i livelli. Naturalmente anche in questo caso vie deve essere una gestione orizzontale dal parte del Project Manager, ma questa risulta di minore entità rispetto a quella verticale. Un ulteriore vantaggio di questo approccio risiede nella possibilità di partizionare facilmente il problema in parti che possono essere sviluppate in parallelo e nel fornire al personale un maggiore coinvolgimento nel progetto.

6 Conclusioni

In questo lavoro sono stati esposti quali sono i problemi dei sistemi di tempo reale e quali sono le principali caratteristiche del paradigma orientato agli oggetti. Nella discussione sono state fatte notare quali sono i problemi che si incontrano nell'adottare tale paradigma per la specifica dei sistemi di tempo reale. Questa discussione è ben lontana da essere esaustiva ma è sufficientemente dettagliata ed ad ampio spettro da poter mettere in grado il lettore di operare maggiori ricerche e di effettuare una prima analisi degli eventuali tool o linguaggi ad oggetti che gli verranno proposti per la specifica di sistemi di tempo reale.

Dalla discussione emerge che il problema non è stato completamente risolto anche perché in generale non è risolto il problema della specifica dei sistemi di tempo reale. In ogni caso gli approcci che offrono una maggiore copertura sono quelli che forniscono un sistema integrato, un CASE tool, che sia basato su solide fondamenta matematiche e che al prezzo di poche restrizioni al paradigma object-oriented permetta di effettuare verifiche di correttezza a vari livelli di astrazione e la validazione finale per mezzo di prove di proprietà e/o per simulazione. Naturalmente in questo contesto la presenza di un'interfaccia utente visuale ed amichevole, la generazione automatica di codice e di documentazione, la possibilità di valutare la qualità della specifica, etc. sono caratteristiche ormai indispensabili.

È nostra opinione che nei prossimi anni vi sarà una notevole crescita delle capacità dei tool. Questa crescita sarà dovuta all'integrazione di tecniche, di specifica, di misura e di validazione, ed

ad una più esatta modellazione del ciclo di vita. I CASE tool dei prossimi anni offriranno una maggiore certezza della soddisfazione dei requisiti.

References

[1] G. A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, London, 1986.

[2] A. J. Alencar and J. A. Goguen, "OOZE: An Object Oriented Z Environment", in *Proc. of European Conference on Object Oriented Programming, ECOOP'91*, pp. 180–199, Springer Verlag, Lecture Notes in Computer Sciences, LNCS n.512, 1991.

[3] T. Andrews and C. Harris, "Combining Language and Database Advances in an Object Oriented Development Environment", in *Proceedings OOPSLA'87, 2nd ACM Conference on Object Oriented Programming Languages and Applications, Springer-Verlag, Berlin*, 1987.

[4] ARTIFEX, "ARTIFEX User's Manual, ver.3.0", tech. rep., ARTIS, Turin, Italy, 1993.

[5] M. Baldassari and G. Bruno, "PROTOB: an Object Oriented Methodology for Developing Discrete Event Dynamic Systems", *Computer Languages*,, Vol. 16, No. 1, pp. 39–63, 1991.

[6] T. E. Bihari and P. Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples", *Computer*, pp. 25–32, Dec. 1992.

[7] B. W. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Software*, Vol. 21, No. 5, pp. 61–72, 1988.

[8] G. Booch, "Object-Oriented Development", *IEEE Transactions on Software Engineering*, Vol. 12, pp. 211–221, Feb. 1986.

[9] G. Booch, *Object-Oriented Design with Application.* California, USA: The Benjamin/Cummings Publishing Company, 1991.

[10] R. Braek and O. Haugen, *Engineering Real Time Systems: An object-oriented methodology using SDL.* New York, London: Prentice hall, 1993.

[11] G. Bruno and G. Marchetto, "Process-translatable Petri nets for the rapid Prototyping of process control systems", *IEEE Transactions on Software Engineering*, Vol. 12, pp. 346–357, Feb. 1986.

[12] G. Bucci, M. Campanai, and P. Nesi, "Tools for Specifying Real-Time Systems", *Journal of Real-Time Systems*, p. in press, 1994.

[13] G. Bucci, M. Campanai, P. Nesi, and M. Traversi, "An Object-Oriented CASE Tool for Reactive System Specification", in *Proc. of 6th International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE)*, (Le CNIT, Paris la Defense, France), 15-19 Nov. 1993.

[14] G. Bucci, M. Campanai, P. Nesi, and M. Traversi, "An Object-Oriented Dual Language for Specifying Reactive Systems", in *Proc. of IEEE International Conference on Requirements Engineering, ICRE'94*, (Colorado Spring, Colorado, USA), 18-22 April 1994.

[15] M. Campanai and P. Nesi, "Supporting Object-Oriented Design with Metrics", in *Proc. of the International Conference on Technology of Object-Oriented languages and Systems, TOOLS europe'94*, (Versailles, France), 7-11 March 1994.

[16] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith, "Object-Z: An Object-Oriented Extension to Z", in *Formal Description Techniques* (S. T. Voung, ed.), Elsevier Science, 1990.

[17] P. Coad and E. Yourdon, *Object-Oriented Analysis.* New Jersey, USA: Yourdon Press, 1991.

[18] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design", *IEEE Transactions on Software Engineering*, Vol. 18, pp. 9–18, Jan. 1992.

[19] B. Cox, "Message/Object Programming: an Evolutionary Change in Programming Technology", *IEEE Software*, Vol. 1, No. 1, pp. 50–61, 1984.

[20] B. Cox, *Object-Oriented programming an evolutionary approach.* Ma.: Addison-Wesley, 1987.

[21] S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming", *ACM Computing Survey*, Vol. 20, No. 1, pp. 29–72, 1988.

[22] A. DelBimbo and P. Nesi, "Blackboard-Based Concurrent Object Recognition Using and Object-Oriented Database", in *Proc. of the IEEE International Phoenix Conference on Computers and Communications, IPCCC'92, Scottsdale, AZ, USA*, pp. 172–180, April 1-3 1992.

[23] T. DeMarco, *Structured Analysis and System Specification.* Yourdon Press, Prentice Hall, 1979.

[24] J. Dick and J. Loubersac, "Integrating Structured and Formal Methods: A Visual Approach to VDM", in *Proc. of 3rd European Software Engineering Conference, ESEC91* (A. vanLamsweerde and A. Fuggetta, eds.), (Milan, Italy), pp. 37–59, Springer Verlag, Lecture Notes in Computer Sciences, LNCS 550, Oct. 1991.

[25] J. Diederich and J. Milton, "Object, Message, and Rules in Database Design", in *Object-Oriented Concepts Databases and Applications* (W. Kim and F. H. Lochovsky, eds.), pp. 177–198, New York, USA: Addison-Wesley Publishing Company, ACM Press, 1989.

[26] E. H. H. Dürr and J. vanKatwijk, "VDM++: A Formal Specification Language for Object-Oriented Designs", in *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 7* (G. Heeg, B. Mugnusson, and B. Meyer, eds.), pp. 63–78, Prentice-Hall, 1992.

[27] C. A. Ellis and S. J. Gibbs, "Active Objects: Realities and Possibilities", in *Object-Oriented Concepts Databases and Applications* (W. Kim and F. H. Lochovsky, eds.), pp. 561–572, New York, USA: Addison-Wesley Publishing Company, ACM Press, 1989.

[28] A. Forin, "Real-Time, UNIX and Mach", tech. rep., School of Computer Science, Carnegie Mellon University, Pittsburg, Pa 15213, USA, 1992.

[29] J. Guttag, "Abstract Data Types and Development of Data Structures", *Communications of the ACM*, Vol. 20, pp. 396–404, June 1977.

[30] D. Harel, "On Visual Formalism", *Communications of the ACM*, Vol. 31, pp. 514–530, May 1988.

[31] D. J. Hatley and I. A. Pirbhai, *Strategies for Real Time System Specification.* New York: Dorset House Publishing, 1987.

[32] HOOD, "An Overview of the HOOD Toolset", tech. rep., Software Sciences, 1988.

[33] Y. Ishikawa, H. Tokuda, and C. W. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints", tech. rep., School of Computer Science, Carnegie Mellon University CMU-CS-90-111, Pitsburg, PA, USA, 15 March 1990.

[34] Y. Ishikawa, H. Tokuda, and C. W. Mercer, "An Object-Oriented Real-Time Programming Language", *Computer*, pp. 66–73, Oct. 1992.

[35] M. A. Jackson, *System Development.* New York, USA: Prentice Hall International, C. A. R. Hoare Series, 1983.

[36] S. Jahanian and D. A. Stuart, "A Method for Verifying Properties of Modechart Specifications", in *Proc. of 9th IEEE Real-Time Systems Symposium*, (Huntsville, Ala., USA), pp. 12–21, IEEE Press., 1988.

[37] C. B. Jones, *Systematic Software Development Using VDM.* Prentice Hall, 1986.

[38] K. Lano, "Z++, An Object-Oriented Extension to Z", in *Proc. of the 4th Annual Z User Meeting* (J. E. Nicholls, ed.), (Oxford, UK), pp. 151–172, Workshop in Computing, Springer Verlag, 1991.

[39] B. Liskov and J. Guttag, *Abstraction Specification in Program Development.* Cambridge, MS, USA: The MIT Press, 1986.

[40] B. Liskov, A. Snyder, R. Atkinson, and G. Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM*, Vol. 20, pp. 564–576, Aug. 1977.

[41] D. Mandrioli, "The Specification of Real-Time Systems: a Logical Object-Oriented Approach", in *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS'92*, 1992.

[42] D. Mandrioli, "The Object-Oriented Specification of Real-Time Systems", in *Tutorial Note of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Europe '93*, (Versailles, France), 8-11 March 1993.

[43] J. Martin and J. Odell, *Object Oriented Analysis and Design.* New Jersey, USA: Prentice-Hall, Englewood Cliffs, 1991.

[44] B. Meyer, "On Formalism in Specifications", *IEEE Software*, pp. 6–26, Jan. 1985.

[45] B. Meyer, *Eiffel: a Language and Environment for Software Engineering.* Prentice-Hall, Englewood Cliffs, 1988.

[46] B. Meyer, *Object-Oriented Software Construction.* New York, USA: Prentice Hall, C. A. R. Hoare Series, 1988.

[47] P. Nesi, "Managing Large Object-Oriented Systems", tech. rep., RT 14/94, Dipartimento di Sistemi e Informatica, Facolta di Ingegneria, Universita delgi Studi di Firenze, Florence, Italy, 1994.

[48] O. Nierstrasz, "A Survey of Object-Oriented Concepts", in *Object-Oriented Concepts Databases and Applications* (W. Kim and F. H. Lochovsky, eds.), pp. 3–22, New York, USA: Addison-Wesley Publishing Company, ACM Press, 1989.

[49] O. M. Nierstrasz, L. Dami, V. deMey, M. Stadelmann, D. Tsichritzis, and J. Vitek, "Visual Scripting: Towards Interactive Construction of Object-Oriented Applications", in *Perspective on Object Oriented Reasearch* (D. Tsichritzis, ed.), 1992.

[50] P. V. Norden, "Curve Fitting for a Model of Applied Research and Development Scheduling", *IBM Journal*, pp. 232–248, July 1958.

[51] NorthernTelecom, "ObjecTime: Object-Oriented CASE for Real-Time Systems", tech. rep., Bell-Northern Telecom, 1993.

[52] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, pp. 1053–1058, Dec. 1972.

[53] P. Pulli, R. Elmstrom, G. Leon, and J. A. delaPuente, "IPTES - Incremental Prototyping Technology for Embedded real-time Systems", in *Proc. of 1991 ESPRIT Conference*, 1991.

[54] A. Rockstrom and R. Saracco, "SDL - CCITT Specification and Description language", *IEEE Transactions on Communications*, Vol. 30, pp. 1310–1318, June 1982.

[55] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* New Jersey: Prentice Hall International, Englewood Cliffs, 1991.

[56] B. Sanden, "Entity-Life Modeling and Structured Analysis in Real-Time Software Design — A Comparison", *Communications of the ACM*, Vol. 32, pp. 1458–1466, Dec. 1989.

[57] B. Sanden, "An Entity-Life Modeling Approach to the Design of Concurrent Software", *Communications of the ACM*, Vol. 32, pp. 330–343, March 1989.

[58] B. Selic, "An Efficient Object-Oriented Variation of Statecharts Formalism for Distributed Real-Time Systems", in *Submitted to CHDL'93: IFIP Conference on Hardware Description language and Their Applications*, (Ottawa, Canada), 26-28 April 1993.

[59] L. Sha and J. B. Goodenough, "Real time Scheduling Theory and Ada", *Computer*, pp. 53–62, April 1990.

[60] S. Shlaer and S. J. Mellor, *Object Oriented Analysis: Modeling the World in Data.* New Jersey, USA: Prentice Hall, Englewood Cliffs, 1988.

[61] S. Shlaer and S. J. Mellor, *Object Life Cycles: Modeling the World in States.* New Jersey, USA: Prentice Hall, Englewood Cliffs, 1991.

[62] A. Snyder, "Inheritance in Object-Oriented Programming Language", in *Inheritance Hierarchies in Knowledge Representation and Programming Languages* (M. Lenzerini, D. Nardi, and M. Simi, eds.), pp. 153–171, Chichester, New York: John Wiley & Sons, 1985.

[63] J. A. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems", *IEEE Computer*, pp. 10–19, Oct. 1988.

[64] J. A. Stankovic and K. Ramamritham, *Advances in Real-Time Systems.* Washington: IEEE Computer Society Press, 1992.

[65] B. Stroustrup, *The C++ Programming Language.* Mass. USA: Addison Wesley Publishing Company, 1986.

[66] B. Stroustrup, "What is Object-Oriented Programming?", *IEEE Software*, pp. 10–20, May 1986.

[67] D. Thomas, "What's in an Object?", *BYTE*, Vol. 14, pp. 231–240, March 1989.

[68] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", in *Proc. Usenix Mach Workshop*, pp. 1–10, Oct. 1990.

[69] C. Tomlinson and M. Scheevel, "Concurrent Object-Oriented Programming Languages", in *Object-Oriented Concepts Databases and Applications* (W. Kim and F. H. Lochovsky, eds.), pp. 79–124, New York, USA: Addison-Wesley Publishing Company, ACM Press, 1989.

[70] I. J. Walker, "Requirements of an Object-Oriented Design Method", *Software Engineering Journal*, pp. 102–113, March 1992.

[71] P. T. Ward and S. J. Mellor, *Structured Development for Real-time Systems.* NJ, USA: Prentice Hall, Englewood Cliffs, 1985.

[72] A. J. Wasserman, P. A. Pircher, and R. J. Muller, "The Object-Oriented Structured Design Notation for Software Design Representation", *Computer*, pp. 50–63, March 1990.

[73] P. Wegner, "The Object-Oriented Classification Paradigm", in *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, eds.), Cambridge MS USA: MIT Press, 1987.

[74] J. M. Wing, "A Specifier's Introduction for Formal Methods", *Computer*, pp. 8–24, Sept. 1990.

[75] R. J. Wirfs-Brock, B. Wilkerson, and L. Winer, *Designing Object Oriented Software.* N.J., USA: Prentice Hall, Englewood Cliffs, 1990.

[76] R. Wirsf-Brock and B. Wilkerson, "Object-Oriented Design: a responsibility-driven approach", in *Proc. OOPSLA'89*, (New Orleans, Louisiana,), pp. 71–75, SIGPLAN NOT, ACM Press, Oct. 1989.

[77] A. Yonezawa and M. Tokoro, *Object-Oriented Concurrent Programming.* Cambridge Massachusetts: The MIT Press, 1987.

[78] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. New Jersey, USA: Prentice-Hall, Englewood Cliffs, 1979.