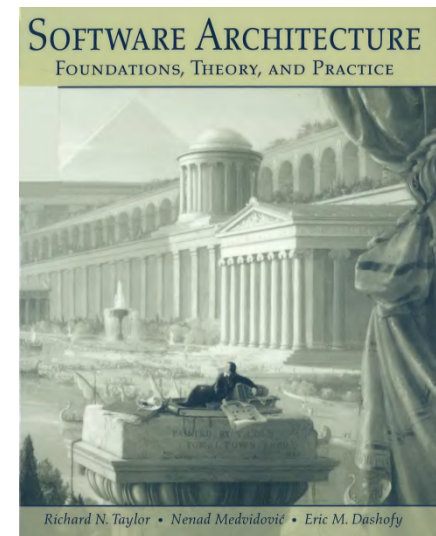


# Stili complessi

Rif. Cap. 4 –Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

**Corso PSSS- Prof. Fasolino – 2020**



# Outline



- Stili più complessi
  - Stile C2 (Componenti e Connettori)
  - Stile a Oggetti distribuiti
- Confronto fra i diversi stili

# Stili Eterogenei

- Molti stili architetturali più complessi si ottengono componendo stili più semplici.
- Es. Stile REST
- • Stile C2- Componenti e Connettori
  - Implicit invocation + Layering + altri vincoli
- • Distributed objects 
  - OO + client-server network style
  - Es. CORBA

# Stile C2- Componenti e Connettori

## Generalità

- Lo Stile C2 nasce dall'idea di usare il pattern **MVC** in una piattaforma distribuita ed eterogenea.
- Furono aggiunti concetti relativi allo **stile a livelli** e **event-based**
- Inizialmente concepito per supportare applicazioni con GUI, è stato poi utilizzato anche in altri tipi di applicazioni.
- Uno stile basato su **invocazione indiretta**, **asincrona**, in cui componenti indipendenti comunicano esclusivamente attraverso **connettori attivi** di message routing (di instradamento messaggi).
- Stringenti regole relative alle connessioni fra componenti che portano ad una architettura a livelli.
- Rif. <http://www.ics.uci.edu/~taylor/documents/1995-C2-TSE.pdf><sup>4</sup>

## Topologia del C2...




- Le applicazioni nello stile C2 sono costituite da **livelli** formati da reti di **componenti concorrenti**, interconnessi mediante **connettori di instradamento di messaggi**.
- C2 è tutto basato **sull'invocazione implicita**. Non c'è nessuna connessione diretta fra componenti.
- Tutti i componenti e connettori hanno un livello "top" e un livello bottom".
  - Il **top** di un componente può connettersi solo al bottom di un solo connettore
  - Il **bottom** di un componente può connettersi solo al top di un solo connettore.
  - Non è permessa altra connessione (per **evitare cicli**)

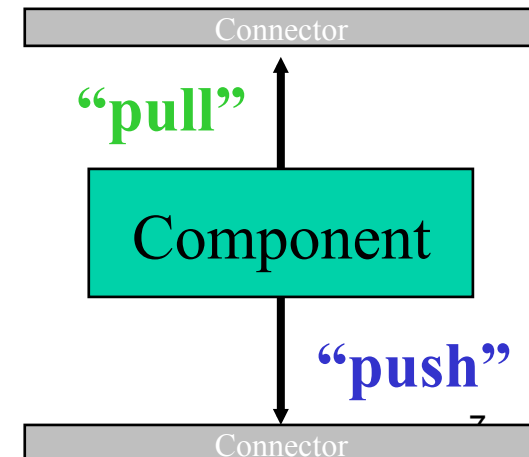
# Topologia del C2

- Un numero arbitrario di componenti e connettori possono essere attaccati a ciascun connettore.
- Due connettori possono anche interconnettersi tra loro, ma sempre rispettando la regola: il lato basso di un connettore- con il lato alto dell'altro.
- Questa regola produce il **layering dell'architettura**.
- Il layering promuove:
  - l'indipendenza dal livello inferiore
  - la sostituibilità dei componenti



# Comunicazione basata su messaggi

- I componenti comunicano attraverso messaggi di due tipi:
- **Richiesta di un Servizio** 
- **Notifiche** (es. notifiche di variazioni di stato di un oggetto) 
  - È coerente con la natura asincrona di applicazioni dove sia gli utenti che l'applicazione svolgono azioni concorrenti in momenti arbitrari, e gli altri componenti ricevono notifica di tali azioni. 
- Le Richieste salgono
- Le notifiche scendono
- Ogni componente è consapevole solo dei componenti sovrastanti
- È indipendente dai livelli inferiori



# Interfaccia del Componente

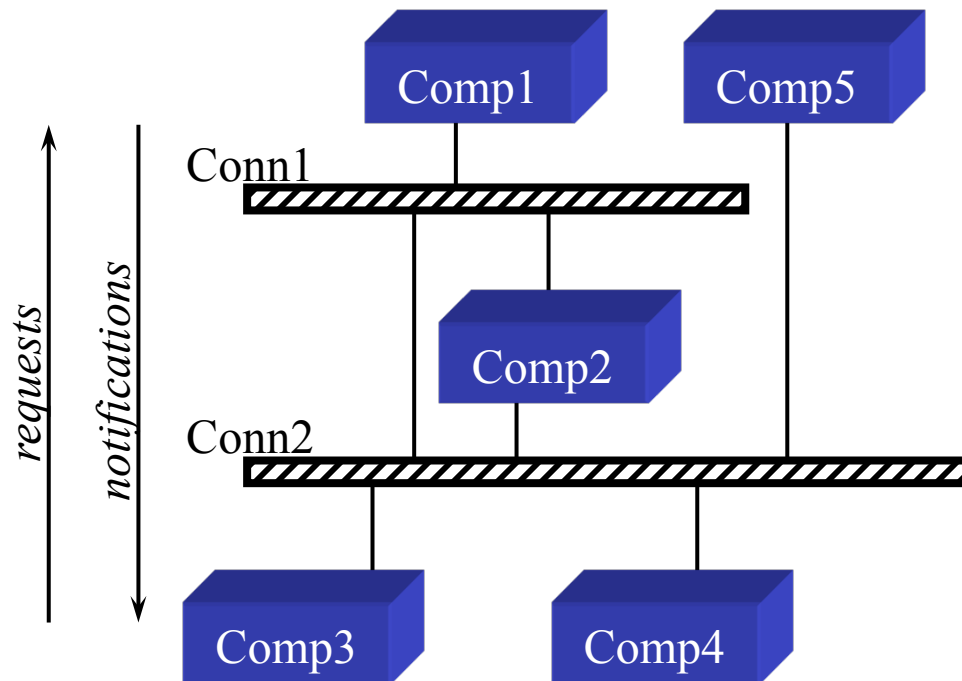


- L'interfaccia avrà un dominio superiore per:
  - Inviare richieste ai componenti superiori
  - Ricevere notifiche dai componenti superiori
- E avrà un dominio inferiore per:
  - Ricevere richieste dai componenti inferiori
  - Inviare notifiche verso i componenti inferiori
- La dipendenza dei componenti dallo strato superiore si può ridurre attraverso connettori che usano meccanismi di traduzione del formato delle richieste e delle notifiche .
  - Il componente non è accoppiato direttamente ai livelli superiori





# Interfacce dei componenti



```
Comp1
  top
  bottom
    request
      query(ID: int, M: msg)
    notification
      ack(ID: int)
```

```
Comp2
  top
    request
    notification
      ack(ID: int)
  bottom
    request
      init(ID: int)
    notification
```

...

## C2 Style (Sintesi)

- **Componenti:** Produttori e/o consumatori di messaggi indipendenti e potenzialmente concorrenti
- **Connettori:** Instradatori di messaggi che possono filtrare, tradurre e inoltrare in broadcast messaggi di due tipi: *notifiche* e *richieste*.
- **Data Elements:** Messaggi – dati spediti attraverso i connettori. I *messaggi di notifica* annunciano i cambiamenti di stato. I *messaggi di richiesta* richiedono l'esecuzione di un servizio.
- **Topology:** Layers di componenti e connettori, ciascuno con un definito dominio “top” e “bottom”:
  - le notifiche vanno verso il basso e le richieste verso l'alto.
  - Ogni componente deve conoscere solo i componenti al di sopra di esso, ed ignorare quelli sottostanti (Indipendenza dal substrato inferiore)

## Benefici dello stile C2

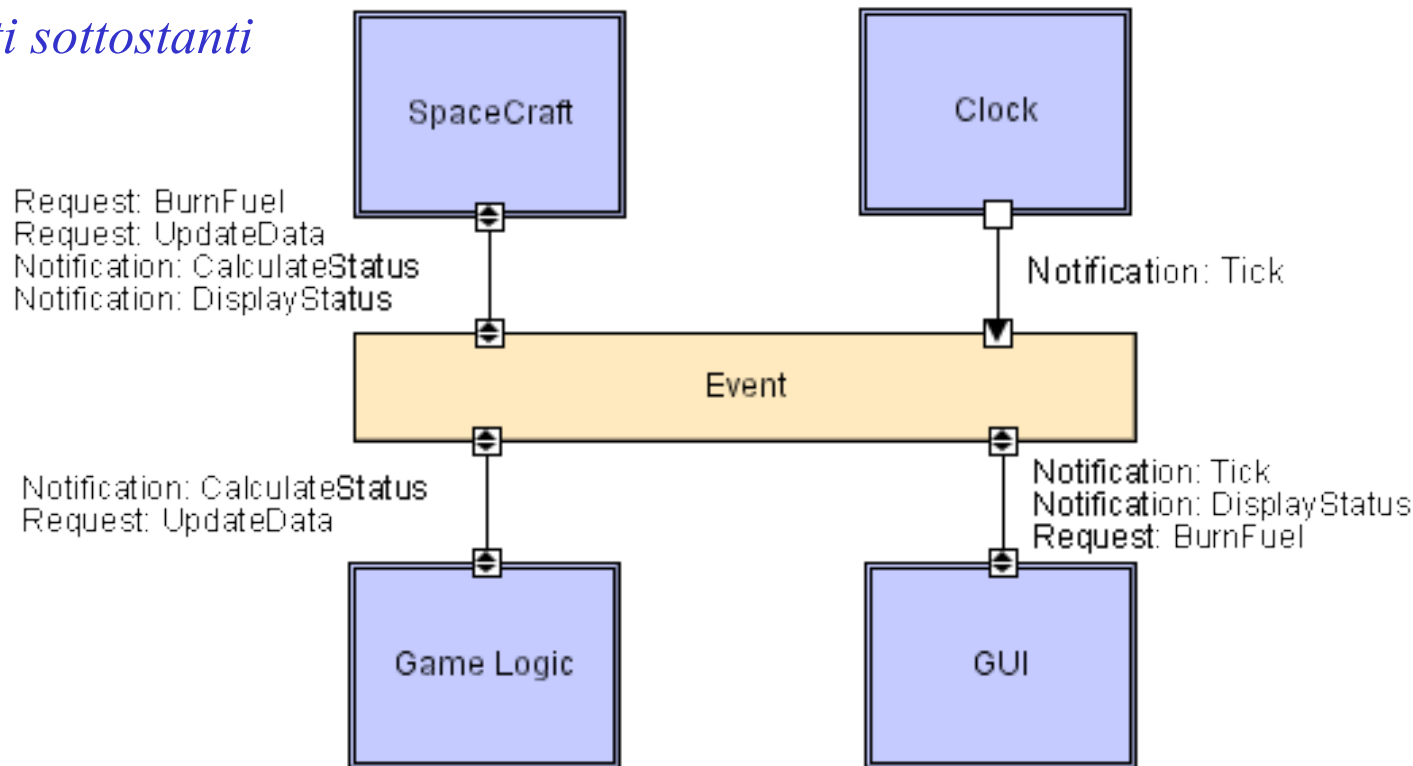
- **Indipendenza dal substrato:**
  - Grazie a ciò, è una architettura facilmente modificabile per adattarsi a funzionare su nuove piattaforme
- **Eterogeneità consentita:**
  - L'applicazione può essere composta da componenti eterogenei, scritti in vari linguaggi e eseguiti su hardware diverso, comunicanti mediante una rete
- **Supporto alle Linee di prodotto**
  - Facile sostituibilità di un componente con un altro, per realizzare applicazioni simili, ma diverse

## Benefici del C2

- Capacità di progettare secondo lo stile MVC
- Supporto naturale a componenti concorrenti
- Supporto per realizzare applicazioni distribuite
  - I dettagli dei protocolli di comunicazione sono lasciati fuori dai componenti e confinati nei connettori
- Usi tipici: applicazioni reattive ed eterogenee, con elevata adattabilità

# C2 per Lunar Lander

*Spacecraft e Clock ignorano  
l'esistenza dei  
componenti sottostanti*



*GameLogic e GUI possono invocare  
i servizi sovrastanti e rispondere alle  
notifiche*

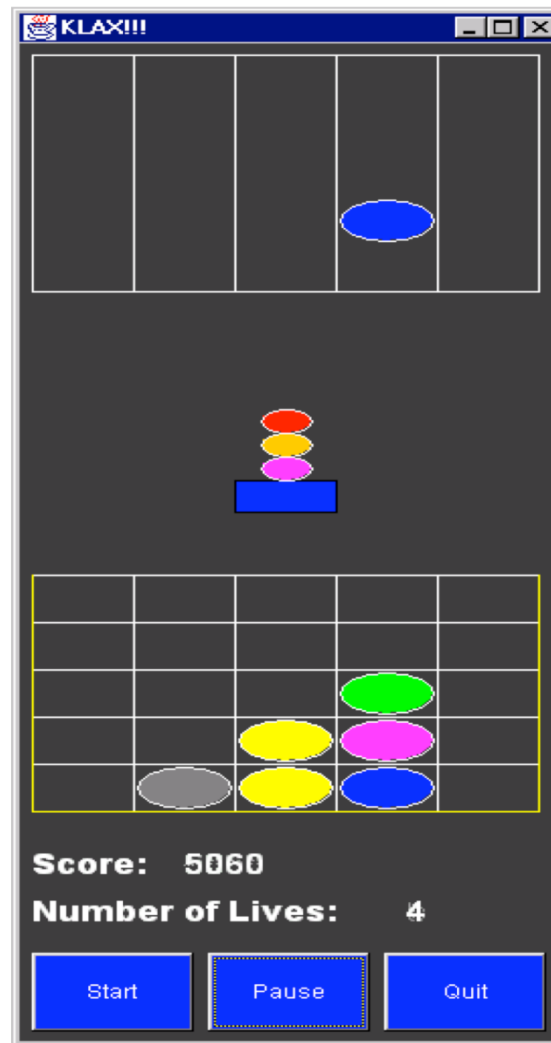
## C2 per Lunar Lander

- In alto i componenti **SpaceCraft** e il **Clock**, che inviano notifiche verso il basso.
- **GameLogic** riceve la notifica dal Clock e richiede al livello superiore di UpDateData.
- **SpaceCraft** riceve la richiesta di UpDateData e ricalcola i dati della navicella. Quindi manda una notifica di CalculateStatus (per la logica del gioco) e DisplayStatus (per la GUI).
- **GameLogic** riceve la notifica di CalculateStatus e rivaluta se il gioco è terminato oppure no.
- GUI manda in alto la richiesta di BurnFuel
- GUI riceve la notifica del Tick per aggiornare il BurnFuel e la notifica di Visualizzare il nuovo Stato del gioco

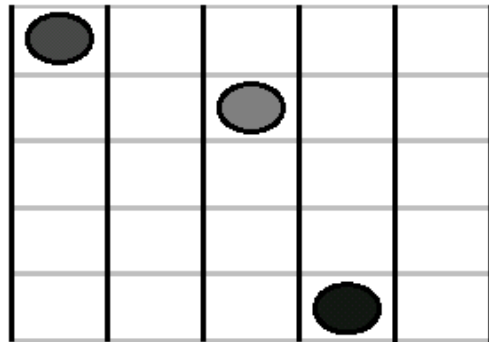
# Esempio: Il Videogioco KLAX

*‘Clone’ del mitico **tetris**, lo scopo è raccogliere sulla paletta i tasselli colorati che cadono dall'alto e depositarli nella zona inferiore per formare dei Klax.*

Un Klax è una riga orizzontale, verticale o obliqua, di almeno 3 tasselli dello stesso colore.



# Esempio: KLAX



## KLAX Chute (Scivolo)

- Tiles of random colors drop one cell at a time, starting at random times and locations

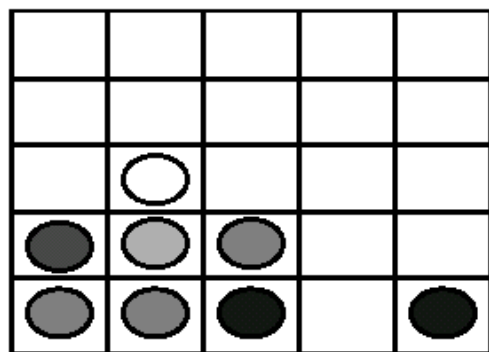
## KLAX Palette

- Palette manipulated to catch tiles coming down the Chute and to drop them into the Well



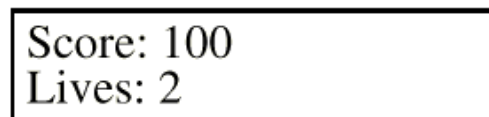
## KLAX Well (pozzo)

- Horizontal, vertical and diagonal sets of three or more consecutive tiles of the same color are removed, and any tiles above them collapse down to fill in the newly-created empty spaces



## KLAX Status

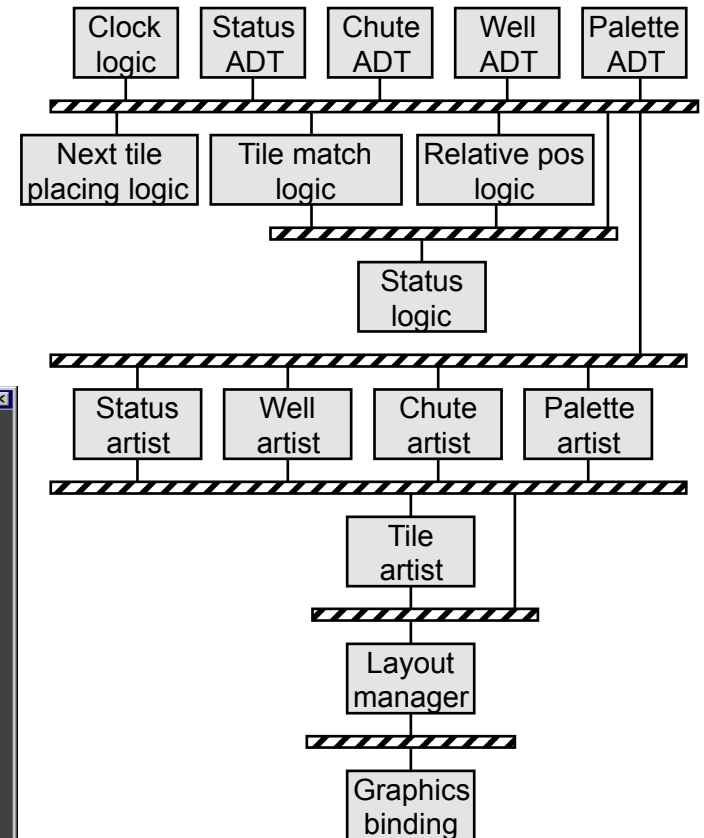
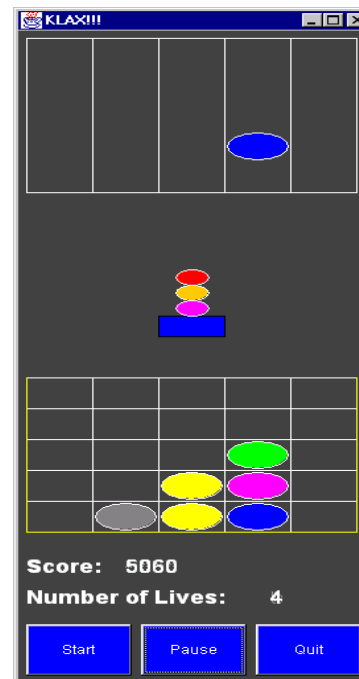
- Points scored as sets are formed
- Lives lost as Well or Palette spills over





# Architettura C2 di KLAX

- KLAX™ game
  - 16 components, approx. 4k SLOC, 100kb compiled
  - implemented from scratch in the C2 architectural style



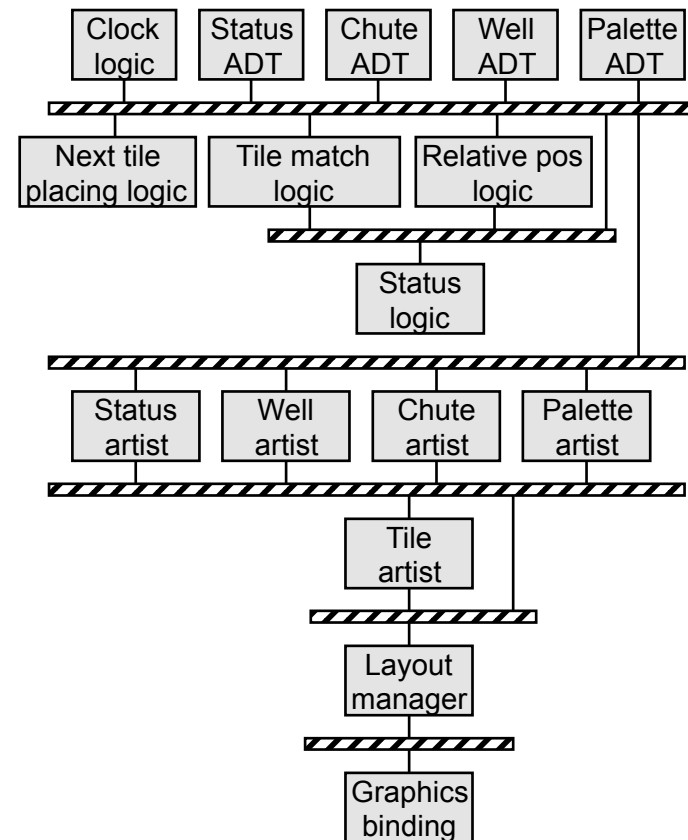
# Architettura del gioco KLAX in C2

Tre gruppi logici di componenti:

-i comp. in alto mantengono **stato del gioco e clock**

-Gli intermedi sono responsabili della **logica del gioco**: richiedono cambiamenti di stato secondo le regole del gioco, e interpretano le notifiche di modifica dello stato del gioco per determinare lo stato del gioco in corso.

-I componenti **artist** ricevono notifiche di cambiamento dello stato del gioco, che causano l'aggiornamento della descrizione dei vari elementi (palette,



Analogia con l'MVC: quando lo stato degli oggetti in alto (ADT) cambia (MODEL), essi mandano le notifiche sia verso i componenti responsabili della logica (CONTROLLER), sia verso i componenti responsabili di mostrare le VIEW (Artist)

## Stile ad Oggetti Distribuiti

# Stile ad Oggetti Distribuiti

- É una combinazione di stile **object oriented** e **client-server** che consente di introdurre il concetto di oggetti distribuiti, accessibili da diversi processi in esecuzione su differenti computers.
- La necessità di comunicazione cross..machine e cross..language obbliga alla serializzazione dei parametri (data marshaling) → (analogie col **pipe&filter**)

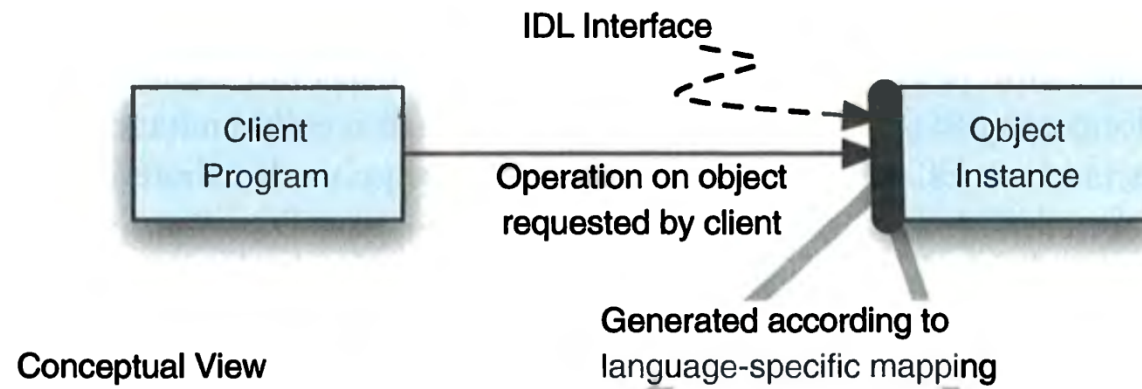
# Distributed Objects: CORBA

- “Oggetti” (a grana grossa o fine) sono in esecuzione su host eterogenei, scritti in linguaggi eterogenei. Forniscono servizi attraverso interfacce ben definite.
- Gli oggetti invocano metodi attraverso i confini di host, processi e linguaggi attraverso **remote procedure calls (RPCs)**.

# Caratteristiche dello stile

- **Componenti:** Oggetti (componenti software che espongono servizi attraverso interfacce )
  - **Connettori:** Invocazione di Metodi remoti (Remote) Method invocation
  - **Dati :** argomenti dei metodi, valori di ritorno, ed eccezioni.
  - **Topologia:** Grafo generico di oggetti, con relazioni dal chiamante al chiamato.
  - Vincoli aggiuntivi: I dati passati attraverso RPC devono essere **serializzabili**. I chiamanti devono gestire le eventuali eccezioni dovute alla rete o errori nei processi.
  - Location, platform, and language “transparency”.
- CAUTION**

# CORBA: richiesta di una operazione su oggetto remoto



# Regole di CORBA

- Le interfacce degli oggetti sono specificate nel linguaggio IDL (Interface Description Language)

```
interface IDataStore{  
    double getAltitude();  
    void setAltitude(in double newAltitude);  
    double getBurnRate();  
    void setBurnRate(in double newBurnRate);  
    void getStatus(out double altitude,  
                   out double burnRate,  
                   out double velocity,  
                   out double fuel,  
                   out double time);  
}
```

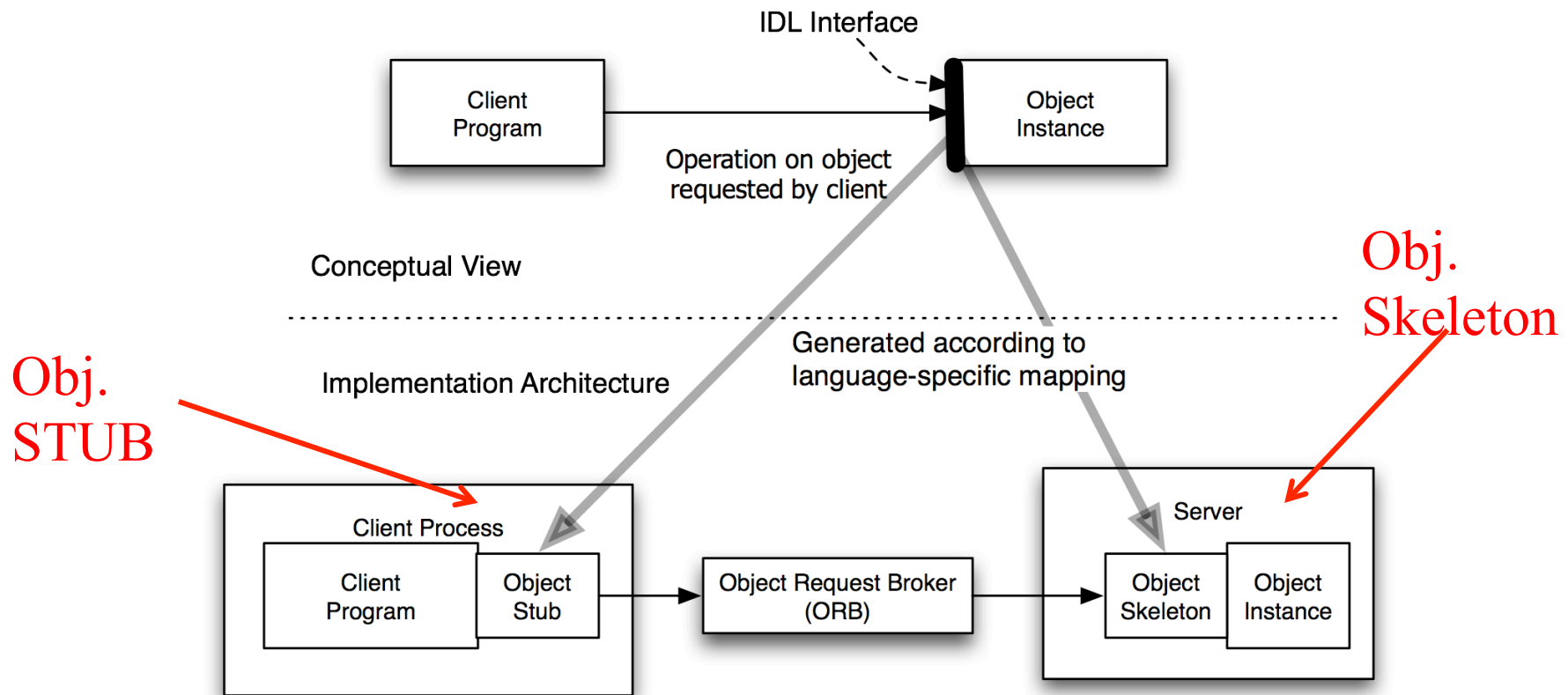
Una interfaccia in IDL

L'accesso ai metodi dell'oggetto avviene mediante chiamate a uno dei metodi dell'interfaccia IDL.

- Il compilatore di IDL** crea due artifatti, **l'object stub** e **l'object skeleton**.



# CORBA Concept and Implementation



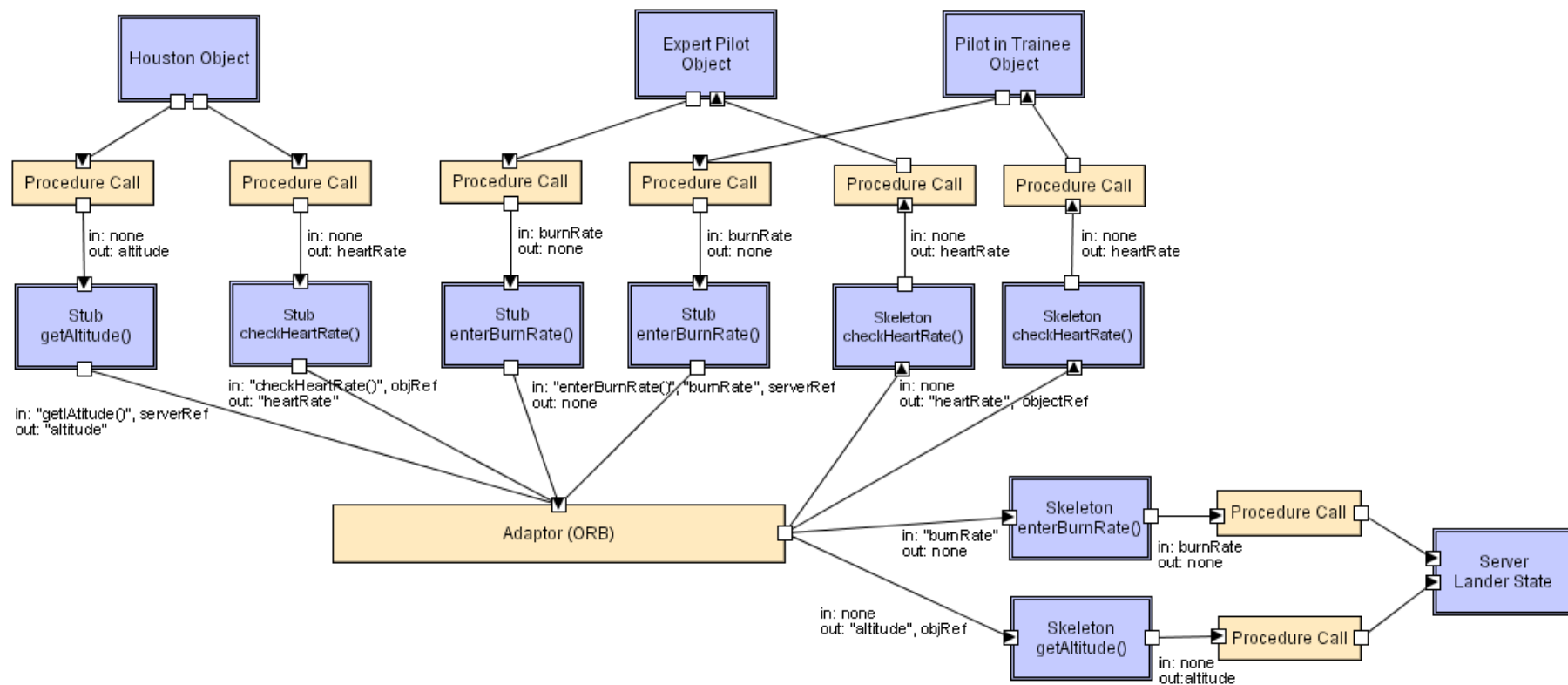
# Object Skeleton

- Uno **object skeleton** è un oggetto nel linguaggio/piattaforma target che fornisce le *implementazioni vuote* per ogni metodo definito nell'interfaccia IDL.
- I tipi dell'IDL sono tradotti in tipi del linguaggio di programmazione locale.
- Lo sviluppatore è quindi responsabile di implementare questi metodi nel linguaggio di programmazione di destinazione. Queste implementazioni costituiscono **l'istanza dell'oggetto**, noto anche come il vero oggetto.

# Object Stub

- **L'object stub** è il complemento dello skeleton: fornisce le implementazioni dei metodi dell'IDL interface, nel linguaggio di programmazione target.
- Tali implementazioni però non forniscono i servizi
- La richiesta dell'oggetto remoto arriva allo stub.
- Lo stub spedisce i parametri della chiamata (trasformati in un blocco dati in binario mediante serializzazione o marshaling) allo skeleton usando la rete o IPC.
- Questa comunicazione è facilitata dall' object request broker, or ORB in CORBA.

# CORBA LL



# Style Summary (1/4)

Style Category & Name	Summary	Use It When	Avoid It When
<b><i>Language-influenced styles</i></b>			
Main Program and Subroutines	Main program controls program execution, calling multiple subroutines.	Application is small and simple.	Complex data structures needed. Future modifications likely.
Object-oriented	Objects encapsulate state and accessing functions	Close mapping between external entities and internal objects is sensible. Many complex and interrelated data structures.	Application is distributed in a heterogeneous network. Strong independence between components necessary. High performance required.
<b><i>Layered</i></b>			
Virtual Machines	Virtual machine, or a layer, offers services to layers above it	Many applications can be based upon a single, common layer of services. Interface service specification resilient when implementation of a layer must change.	Many levels are required (causes inefficiency). Data structures must be accessed from multiple layers.
Client-server	Clients request service from a server	Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation.	Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine capabilities rival or exceed the server's.

## Style Summary, continued (2/4)

### ***Data-flow styles***

Batch sequential	Separate programs executed sequentially, with batched input	Problem easily formulated as a set of sequential, severable steps.	Interactivity or concurrency between components necessary or desirable. Random-access to data required.
Pipe-and-filter	Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters	[As with batch-sequential] Filters are useful in more than one application. Data structures easily serializable.	Interaction between components required. Exchange of complex data structures between components required.

### ***Shared memory***

Blackboard	Independent programs, access and communicate exclusively through a global repository known as blackboard	All calculation centers on a common, changing data structure; Order of processing dynamically determined and data-driven.	Programs deal with independent parts of the common data. Interface to common data susceptible to change. When interactions between the independent programs require complex regulation.
Rule-based	Use facts or rules entered into the knowledge base to resolve a query	Problem data and queries expressible as simple rules over which inference may be performed.	Number of rules is large. Interaction between rules present. High-performance required.

## Style Summary, continued (3/4)

### ***Interpreter***

Interpreter	Interpreter parses and executes the input stream, updating the state maintained by the interpreter	Highly dynamic behavior required. High degree of end-user customizability.	High performance required.
Mobile Code	Code is mobile, that is, it is executed in a remote host	When it is more efficient to move processing to a data set than the data set to processing. When it is desirous to dynamically customize a local processing node through inclusion of external code	Security of mobile code cannot be assured, or sandboxed. When tight control of versions of deployed software is required.

# Style Summary, continued (4/4)

## ***Implicit Invocation***

Publish-subscribe	Publishers broadcast messages to subscribers	Components are very loosely coupled. Subscription data is small and efficiently transported.	When middleware to support high-volume data is unavailable.
Event-based	Independent components asynchronously emit and receive events communicated over event buses	Components are concurrent and independent. Components heterogeneous and network-distributed.	Guarantees on real-time processing of events is required.
<b><i>Peer-to-peer</i></b>	Peers hold state and behavior and can act as both clients and servers	Peers are distributed in a network, can be heterogeneous, and mutually independent. Robust in face of independent failures. Highly scalable.	Trustworthiness of independent peers cannot be assured or managed. Resource discovery inefficient without designated nodes.

## ***More complex styles***

C2	Layered network of concurrent components communicating by events	When independence from substrate technologies required. Heterogeneous applications. When support for product-lines desired.	When high-performance across many layers required. When multiple threads are inefficient.
Distributed Objects	Objects instantiated on different hosts	Objective is to preserve illusion of location-transparency	When high overhead of supporting middleware is excessive. When network properties are unmaskable, in practical terms.