Progettazione avanzata di software di controllo industriale

Elettric80

Programma

Argomento	N. di ore	Data
Design Top Down e Bottom Up	6	06/11/2017
Domain Driven Design	6	13/11/2017
Modelli “Message Passing”	6	20/11/2017
Sistemi concentrati e distribuiti	6	27/11/2017
Processi di sviluppo ed integrazione	6	04/12/2017
Sintesi del caso di studio affrontato	6	11/12/2017

BOOKS

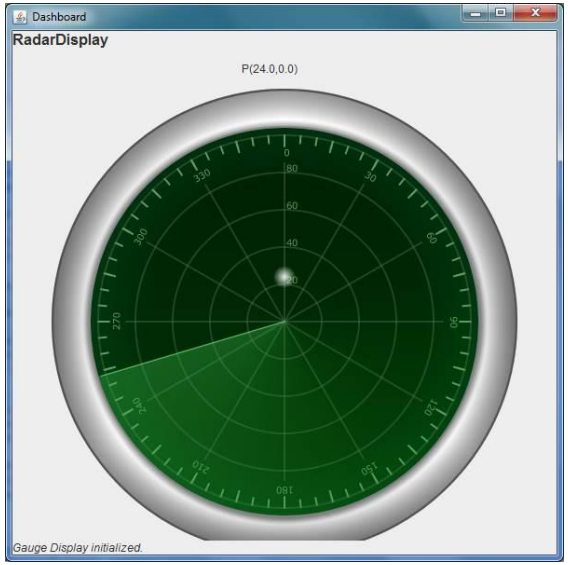
- Domain Driven Design. E. Evans. Addison-Wesley 2004. ISBN 0321125215 978-0321125217
- Building Microservices. S. Newman. O'Reilly 2016. ISBN 978-1491950357
- Implementing Domain-Driven Design. Vaughn Vernon. Addison Wesley. ISBN-10: 0321834577
- Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Gregor Hohpe, Bobby Woolf. ISBN-10: 0133065103
- Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka.Vaughn Addison-Wesley 2016. Vernon.ISBN-10: 0133846830
- Building the Web of Things: With Examples in Node.js and Raspberry Pi. Dominique Guinard, Vlad M. Trifa, Manning 2016, ISBN-10: 1617292680
- Domain Specific Languages. M. Fowler. Addison Wesley. ISBN-10: 0321712943

Il software …

- richiede computer …
  - ma Computer science is no more about computers than astronomy is about telescopes. (**Dijkstra)**
- richiede linguaggi di programmazione …
  - ma perché tanti linguaggi diversi?
  - quale linguaggio usare?
- richiede infrastrutture/frameworks …
  - ma quale impatto sulla analisi dei problemi e sulla progettazione ?
- richiede processi di sviluppo agili …
  - ma come impostarli e utilizzarli al meglio ?
- Una mappa di riferimento

CaseStudy1

Si vuole realizzare un sistema software capace di presentare su un personal computer convenzionale l'immagine grafica di uno schermo radar (detto *radarGui*)



Il sistema deve fare in modo che su *radarGui* venga visualizzata l'informazione acquisita da un sonar HC-SR04 connesso a un RaspberryPi o a un Arduinio.

L'immagine risultante potrebbe essere del tipo mostrato nella figura in cui viene visualizzato un punto a distanza 24 cm in direzione 0.0.
radarGui deve permettere di visualizzare punti a distanza compresa tra cm 0 e cm 80 lungo direzioni comprese tra 0 e 180, utilizzando il quadrante di destra della figura.

PRIMA ANALISI : si tratta di realizzare un sistema software distribuito ed eterogeneo

Sistemi software

Diversamente da un algoritmo, un ***sistema software*** è caratterizzato dalle seguenti proprietà:

- è composto da molteplici entità che interagiscono tra loro
- ha un funzionamento osservabile descritto da "storie di interazione" che specificano i "messaggi" ricevuti e trasmessi;
- il sistema "calcola" attraverso un pattern di interazioni (***interaction history***) iniziate dall'esterno, spesso al di fuori del controllo del sistema stesso;
- opera on-line ed è aperto;
- possiede sia proprietà trasformazionali sia proprietà temporali;
- ha come requisito primario il costo del ciclo di vita;

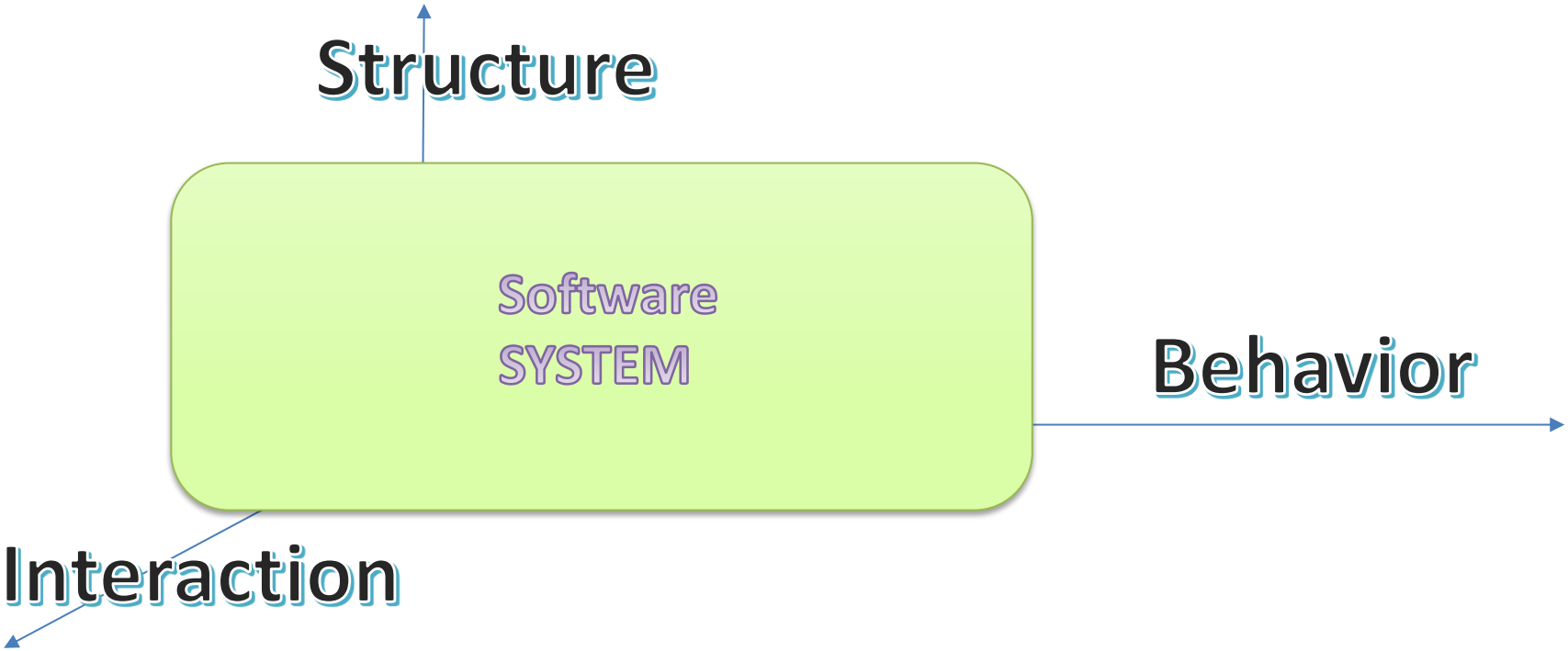
Architetture software

- L'***Open Group Architectural Framework*** definisce architettura: ***"a set of elements depicted in an architectural model and a specification of how these elements are connected to meet the overall requirements of an information system"***.
- *The structure of the components of a system, their interrelationships, and principles and guidelines governing their design and evolution over time (*Garlan and Perry, 1995)
- ***The fundamental organization of a system embodied in its components their relationships to each other and to the environment, and the principles guiding its design and evaluation"***. *(*ANSI/IEEE Std 1471 version from 2000 (adopted as ISO/IEC 42010 in 2007)
- Tra le altre accezioni possibili, una delle più curiose, su cui vale la pena di riflettere, è quella per cui ***l'architettura è ciò che rimane di un sistema quando non si può più togliere nulla, continuando a comprenderne la struttura e il funzionamento***.
- *Design **decisions** about any system that keep implementors and maintainers from exercising **needless creativity** (Gregory Hohope)*

Design pattern

- Le prime esperienze collettive nello studio delle architetture software possono essere fatte risalire al workshop OOPSLA del 1981 guidato da Bruce Anderson che mirava allo sviluppo di un "architecture handbook".
- A questo periodo può anche essere fatto risalire l'idea di pattern culminata nella pubblicazione nel 1995 dell'ormai famoso testo sui ***Design Pattern*** [GHJV95] della così detta GoF (***Gang-of-Four***: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides).
- Da allora si sono susseguiti molte altre conferenze e lavori. I riferimenti più noti sono i cinque testi sulle ***Patten Software Architectures*** ([POSA1], [POSA2], [POSA3], [POSA4], [POSA5]) e i convegni PLoP (***Pattern Languages of Programming***).

Dimensioni



L'attenzione si focalizza su almeno tre diversi punti di vista:

- l'organizzazione del sistema in parti (**struttura**);
- il modo in cui le diverse parti scambiano informazione implicita o esplicita tra loro (**interazione**);
- il funzionamento del tutto e di ogni singola parte (**comportamento**).

Approcci allo sviluppo

BOTTOM-UP

- Parto da una '**tecnologia**'
  - Oo (C++, C#, Java, …)
  - Funzionale (JavaScript, Node)
  - Logica (Prolog)
  - DataBase (SQL, …)
  - Web (REST)
  - …
- Scrivo codice
- Faccio testing
- Faccio deployment

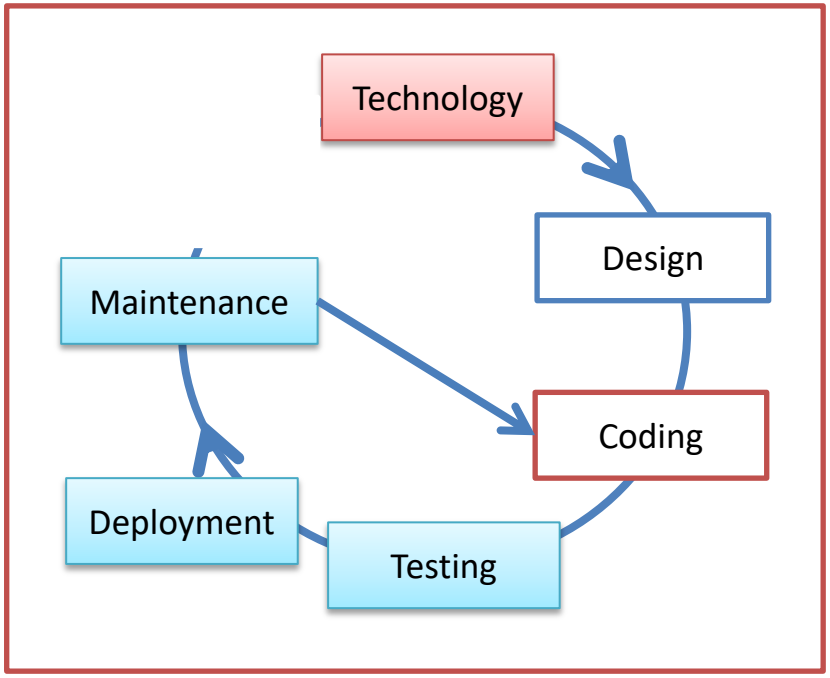
TOP-DOWN

- Parto dal **problema**
  - Analisi dei requisiti
  - Analisi del problema
  - Progetto di soluzione
- Per ogni fase produco un modello
- Implemento il progetto
- Faccio testing
- Faccio deployment

Sviluppo di sistemi sofwtare

PROCESSI BOTTOM-UP

Tecnhology-first



Bottom-up

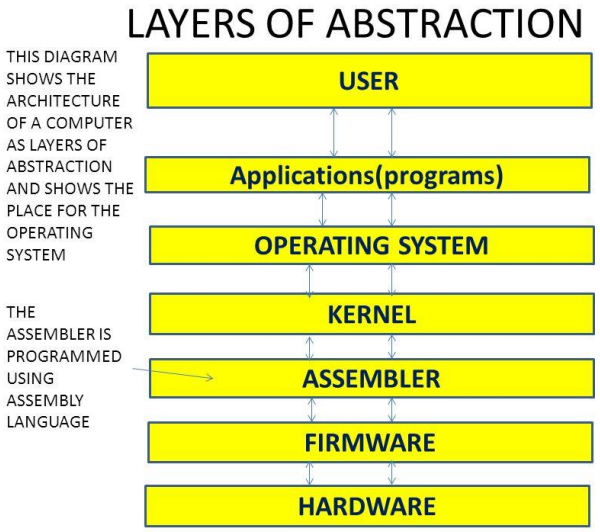
In mathematical logic and theoretical computer science a **register machine** is a generic class of abstract machines used in a manner similar to a Turing machine.

The Minsky machine

ZERO cell
INC cell
SUBJZ cell label
HALT

Is Turing equiivalent (two tapes and a simple Gödelization).

The canonical reference is Minsky's book, *Computation: Finite and Infinite Machines* (Prentice-Hall International, 1967; ISBN 0131655639), in which he calls these machines *program machines*.



Language

- Java (C#, C++ …)
- JavaScript, Node
- Python, Lua ,…
- Koplin
- …

Paradigm /Style

- Oop
- Event driven
- Client-Server
- Actors
- Agents
- …

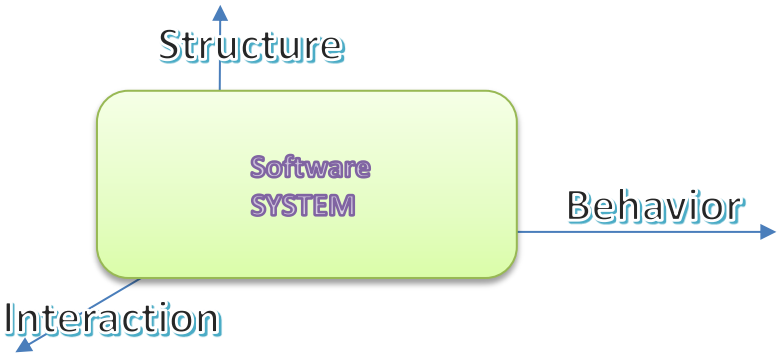
Motti

- *A design without specifications cannot be right or wrong, it can only be surprising!* Young et al. (1985)
- A feature does not exist unless a test validates that it functions.
- *Analyze a little. Design a little. Code a little. Test what you can.*
- Software entities should be open for extension but closed for modification
- *Qualsiasi tecnlogia sufficientemente avanzata é indistinguibile dalla magia (A.L.Clarck)*

Concetti computazionali ‘sistemistici’ di base

- Dati
- Funzioni
- Procedure
- Eventi
- Oggetti
- Messaggi
- Attori
- Agenti
- …

CaseStudy1: design

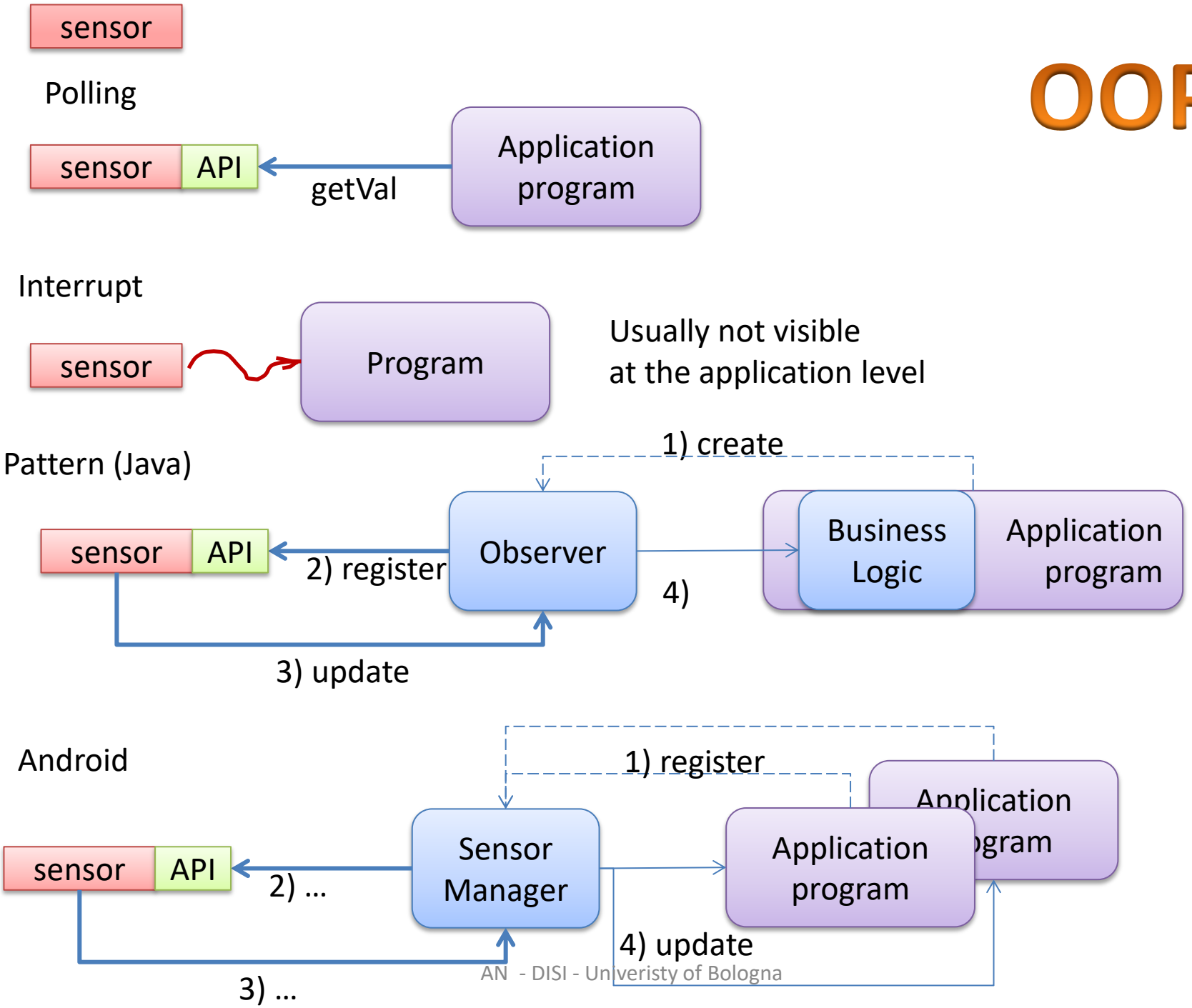


- STRUTTURA: sonar (un sensore) e radarGui (un sottosistema)
- INTERAZIONE: occorre che i dati vadano (via rete) dal sensore al radarGui

PARTIAMO BOTTM-UP : acquisire i dati dal sensore

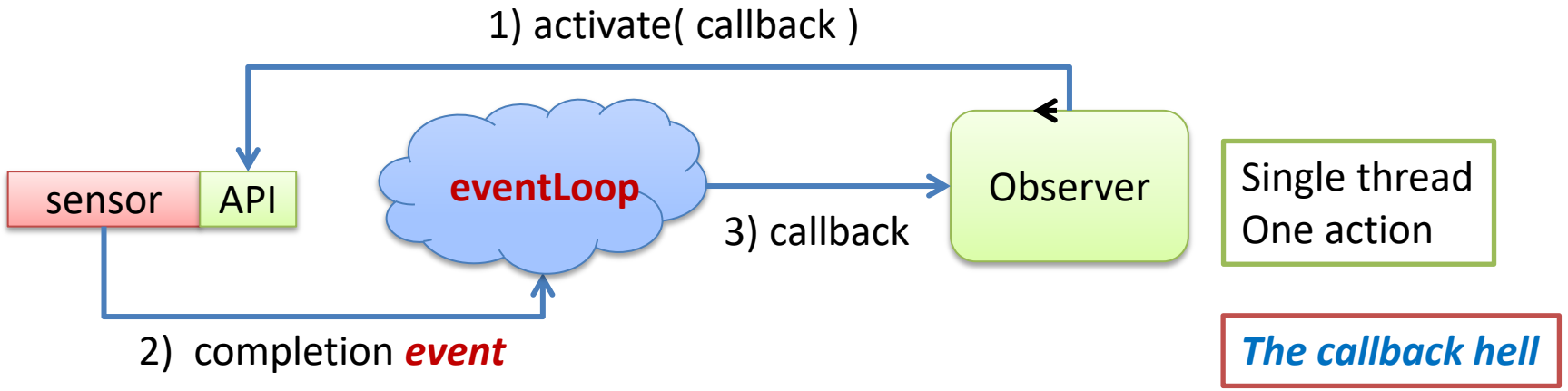
LAVOREREMO MOLTO SUL TEMA-CHIAVE DELLA INTERAZIONE

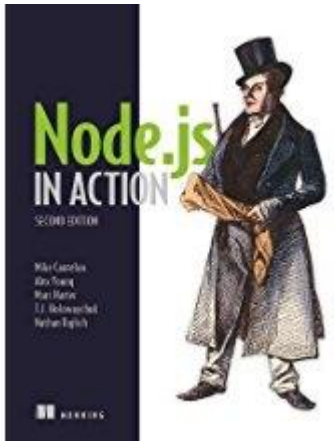
OOP


sensor
Polling
sensor
API
getVal
Application program
Interrupt
sensor
Program
Usually not visible at the application level
Pattern (Java)
1) create
sensor
API
2) register
Observer
4)
Business Logic
Application program
3) update
Android
1) register
sensor
API
2) …
Sensor Manager
Application program
Application
ogram
4) update
3) …

EventDriven

Node (javascript) / **Asynchronous operations**

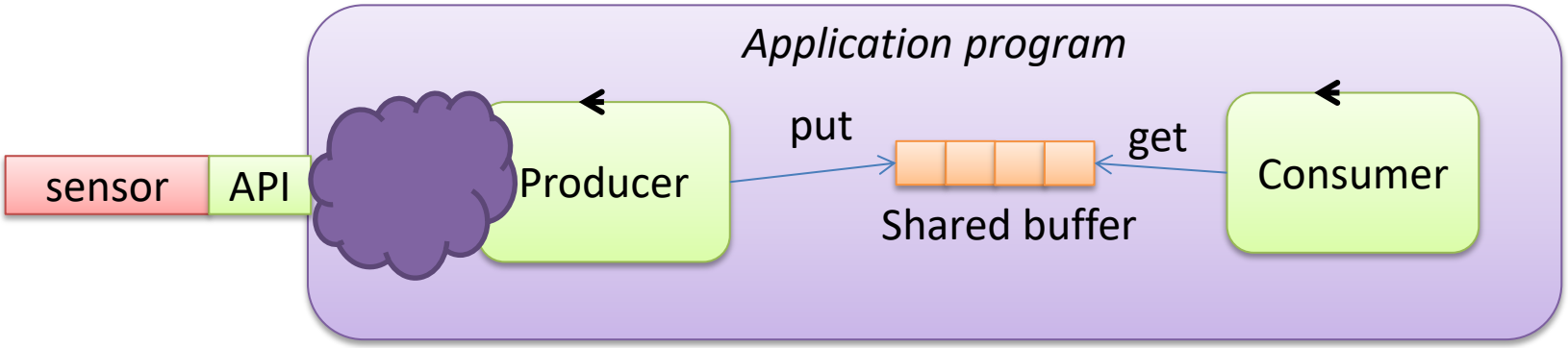




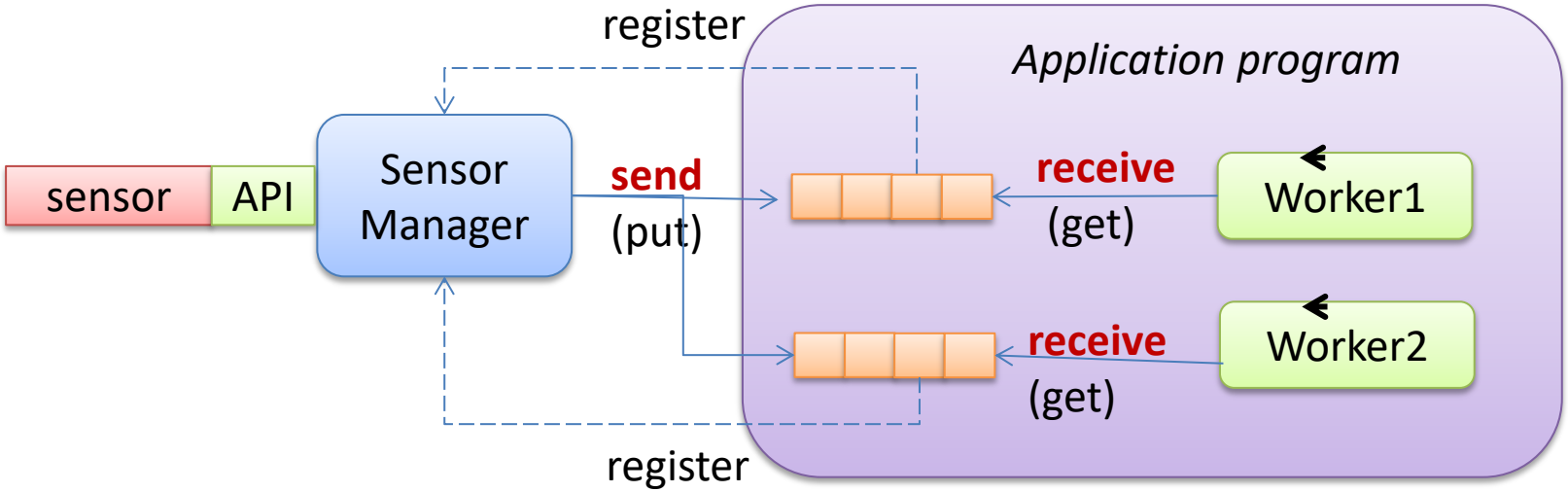
You already know JavaScript. The trick to mastering Node.js is learning how to build applications that fully exploit its powerful asynchronous event handling and non-blocking I/O features. The Node server radically simplifies event-driven real-time apps like chat, games, and live data analytics, and with its incredibly rich ecosystem of modules, tools, and libraries,

OOP: shared buffer

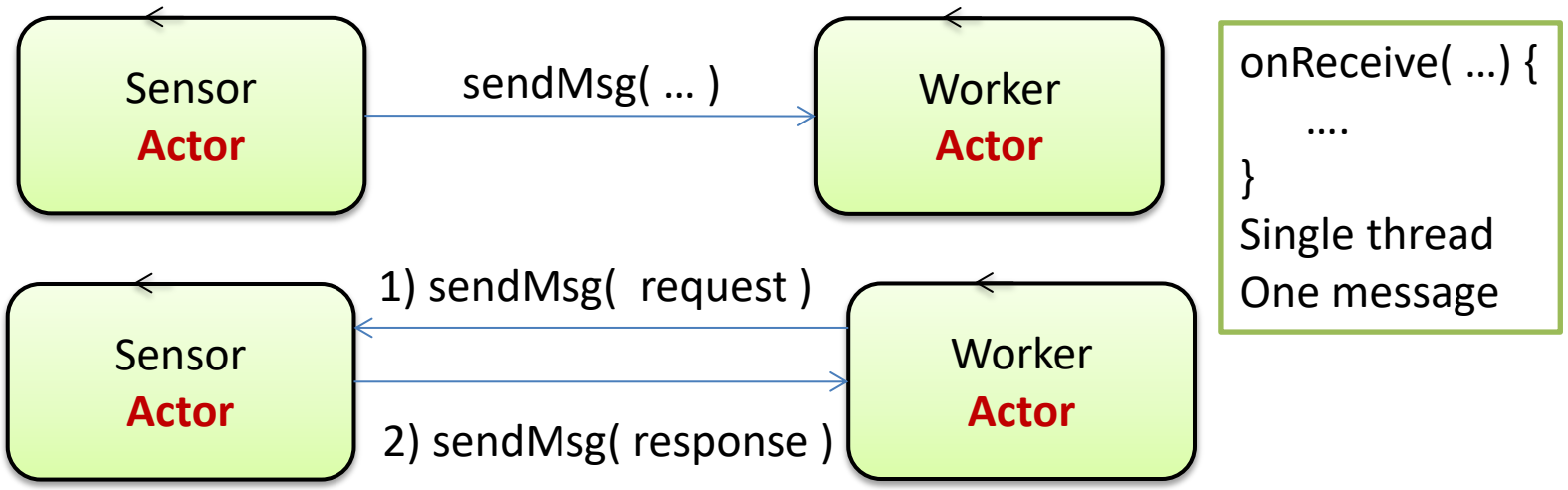
Producer / Consumer

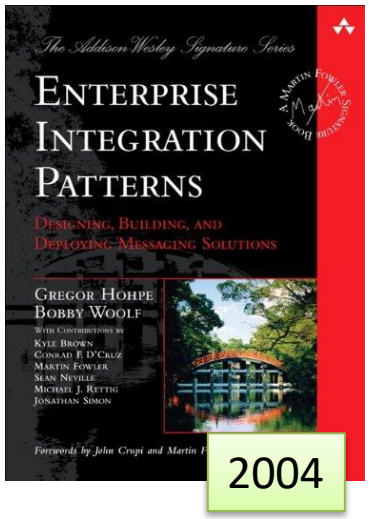


Worker-specific (input) queues



Message Passing





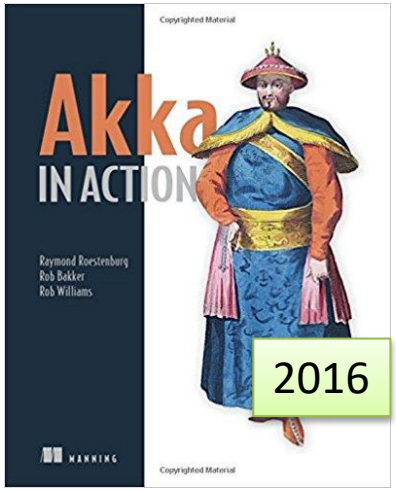
This book is about how to use messages to **integrate applications.** This book provides a **consistent vocabulary** and visual notation framework to describe large-scale integration solutions across **many technologies**. It also explores in detail the advantages and limitations of asynchronous messaging architectures.

The authors also include examples covering a variety of different integration technologies, such as JMS, MSMQ, TIBCO ActiveEnterprise, Microsoft BizTalk, SOAP, and XSL. A case study describing a bond trading system illustrates the patterns in practice, and the book offers a look at emerging standards, as well as insights into what the future of enterprise integration might hold.

Actors

The **actor model** in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation. In response to a message that it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify private state, but can only affect each other through messages (avoiding the need for any locks).

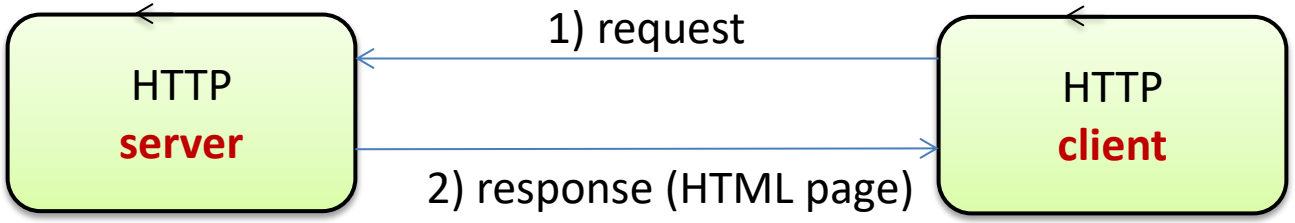
According to Carl Hewitt, unlike previous models of computation, the actor model was inspired by physics, including general relativity and quantum mechanics.
It was also influenced by the programming languages Lisp, Simula and early versions of Smalltalk, as well as capability-based systems and packet switching. Its development was "motivated by the prospect of highly parallel computing machines consisting of many independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network



Akka in Action is a comprehensive tutorial on building message-oriented systems using Akka. The book takes a hands-on approach, where each new concept is followed by an example that shows how it works, how to implement the code, and how to test it.

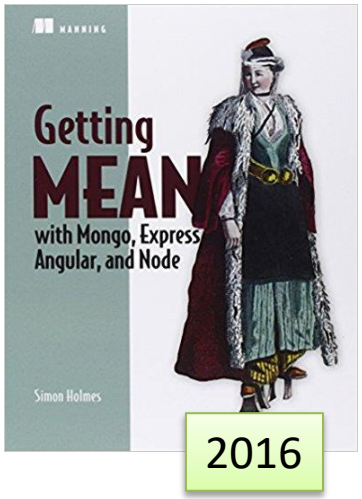
Web

HTTP



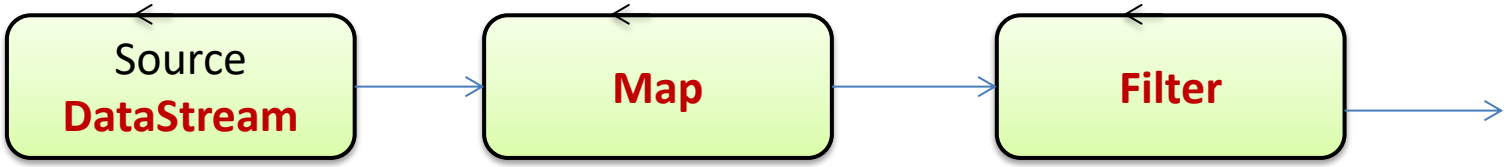
REST

GET	request
POST	create
PUT	update
DELETE	remove
HATEOAS	*hypermedia as the engine of application state*

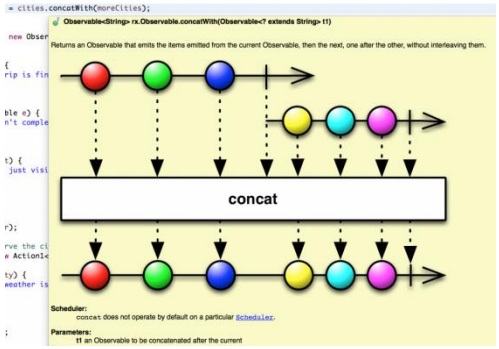


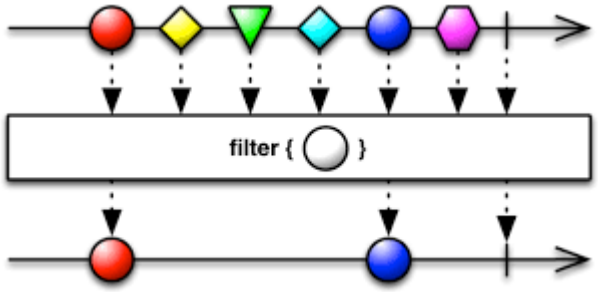
Together, the **MongoDB** database, the **Express** and **AngularJS** web application frameworks, and **Node.js** on the server-side constitute the **MEAN** stack, a powerful web development platform that uses JavaScript top to bottom.

Ansynch / Functional

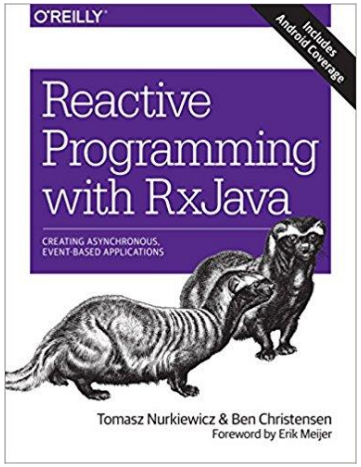


RxJava



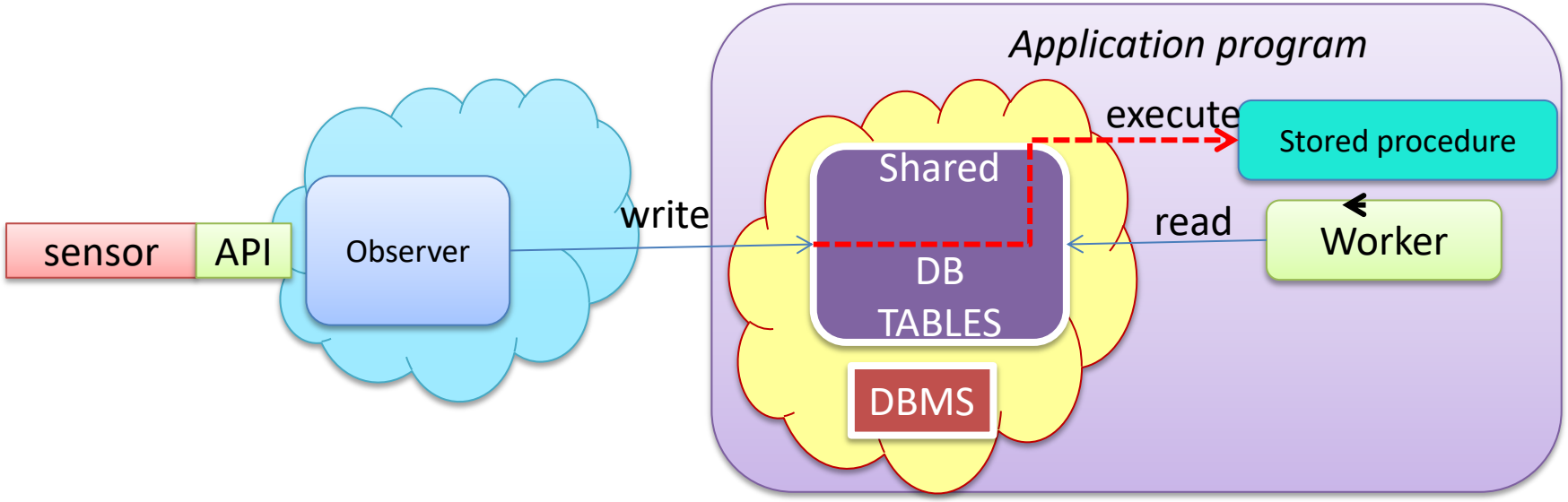


Reactive Streams have been incorporated into the JDK as java.util.concurrent.Flow in version 9.



In today's app-driven era, when programs are asynchronous and responsiveness is so vital, reactive programming can help you write code that's more reliable, easier to scale, and better-performing. With this practical book, Java developers will first learn how to view problems in the reactive way, and then build programs that leverage the best features of this exciting new programming paradigm.

Data base

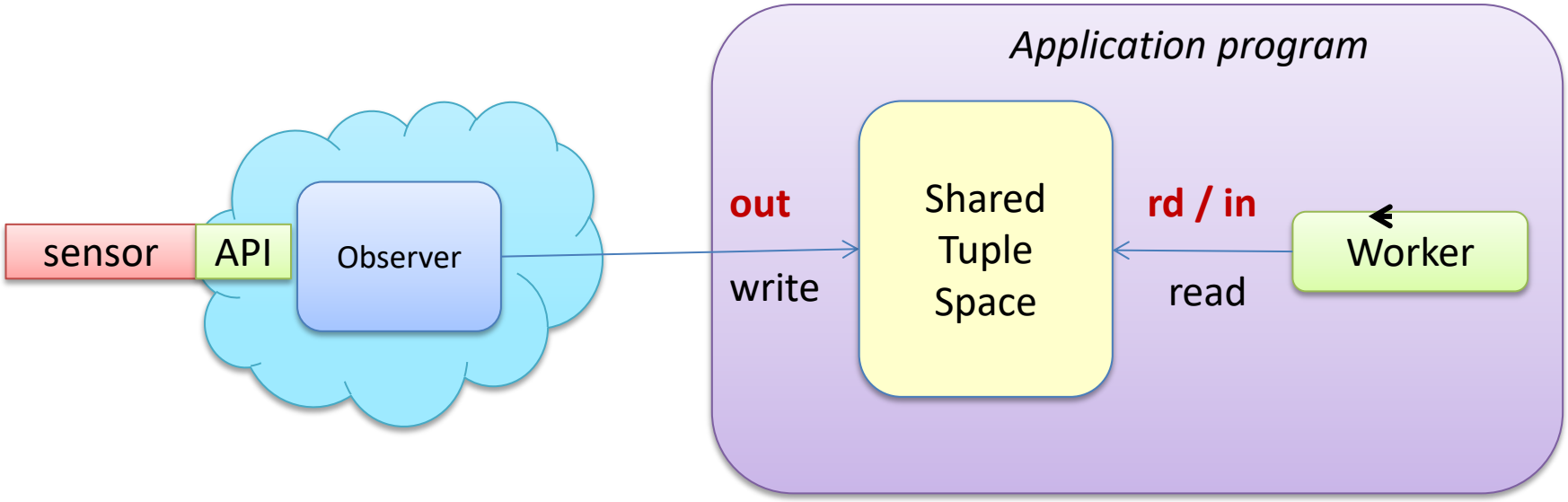


Transactions assure **ACID** properties:

Atomicity, Consistency, Isolation, Durability

https://en.wikipedia.org/wiki/**Distributed_transaction**

Tuple spaces

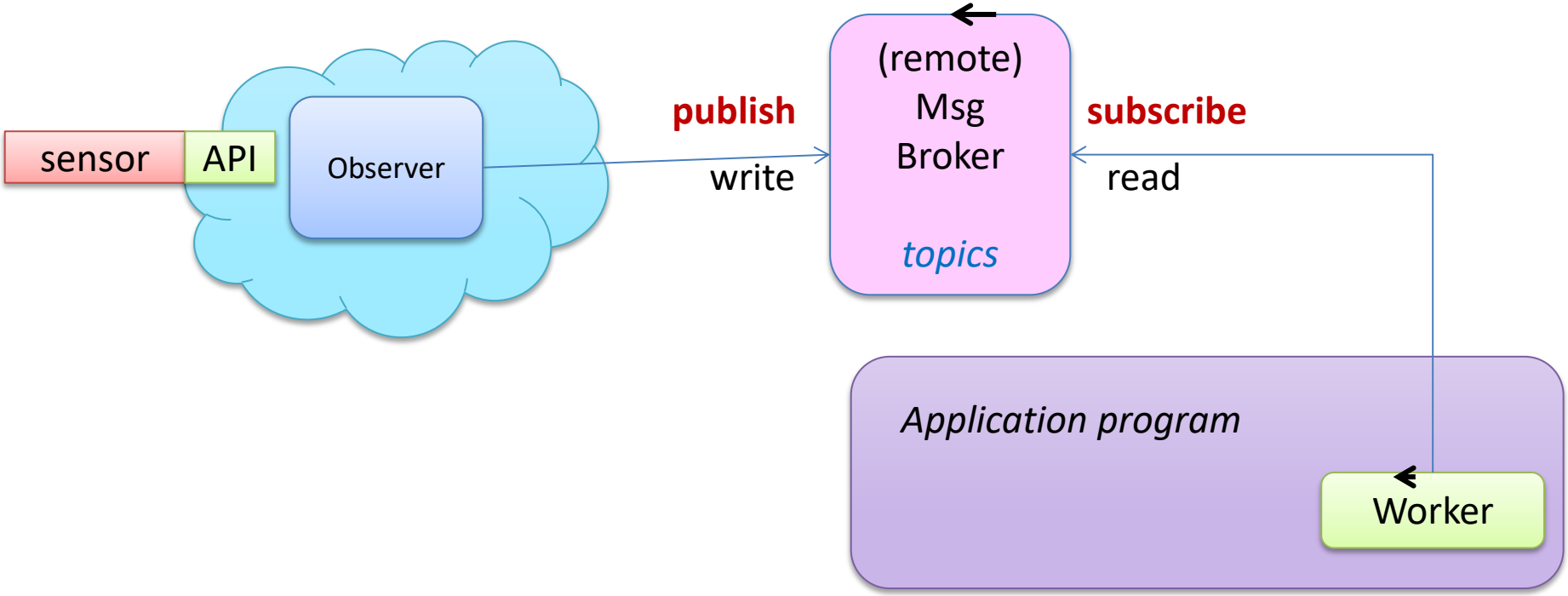


A **tuple space** is an implementation of the associative memory paradigm for parallel/distributed computing.
It provides a repository of tuples that can be accessed concurrently.

Tuple spaces were the theoretical underpinning of the Linda language developed by David Gelernter and Nicholas Carriero at Yale University.
Implementations of tuple spaces have also been developed for Java (JavaSpaces), Lisp, Lua, Prolog, Python, Ruby, Smalltalk, Tcl, and the .NET framework.

MOM

Publish–subscribe pattern



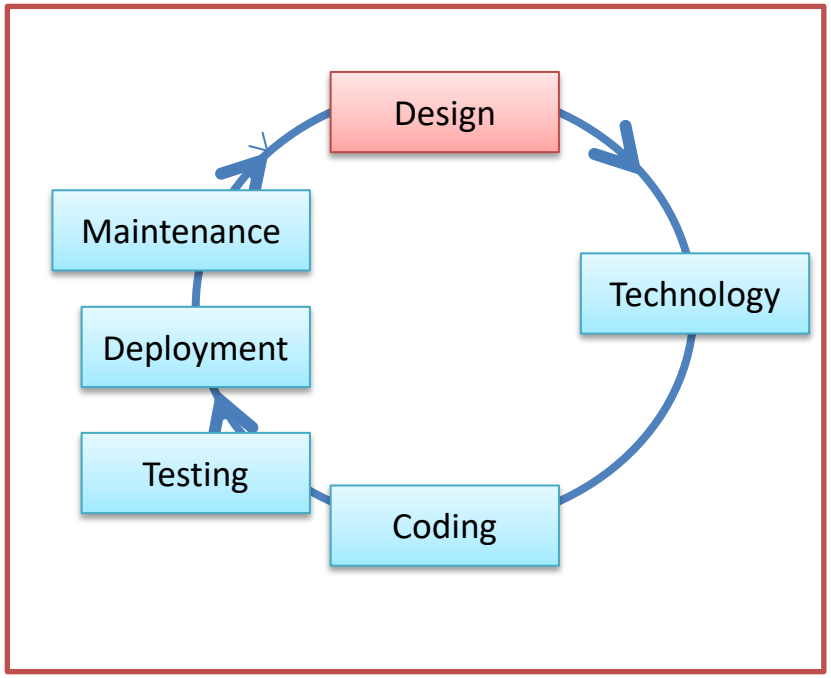
The MQ Telemetry Transport (MQTT) is an ISO standard (ISO/IEC PRF 20922) supported by the OASIS organization.
Eclipse Mosquitto™ is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 3.1 and 3.1.1.

Sviluppo di sistemi sofwtare

PROCESSI TOP-DOWN

Top-down

Model-based



Thre is no code without design

Model of the system
(DESIGN)

HOW

There is no design without problem analysis

Model of the system
(ANALYSIS)

WHAT
HOW

There is no problem without requirements

WHAT

iterate

Model of the system
Requirements

https://en.wikipedia.org/wiki/**Requirements_analysis**

Modelli

- Nel linguaggio comune il termine ***modello*** è spesso usato per denotare un'astrazione di qualcosa che esiste nella realtà, come ad esempio il modello che posa per un artista, una riproduzione in miniatura, un esempio di modo di svolgere un'attività, una forma da cui ricavare vestiti, un ideale da seguire, etc..
- Alcuni (tra cui gli ingegneri) intendono per ***modello*** un sistema matematico o fisico che ubbidisce a specifici vincoli e che può essere utilizzato per descrivere e comprendere un sistema (fisico, biologico, sociale, etc.) attraverso relazioni di analogia.

Modelli nel software

Nel contesto dei processi di costruzione del software, il termine **modello** va primariamente inteso come ***un insieme di concetti e proprietà volti a catturare aspetti essenziali di un sistema, collocandosi in un preciso spazio concettuale***.

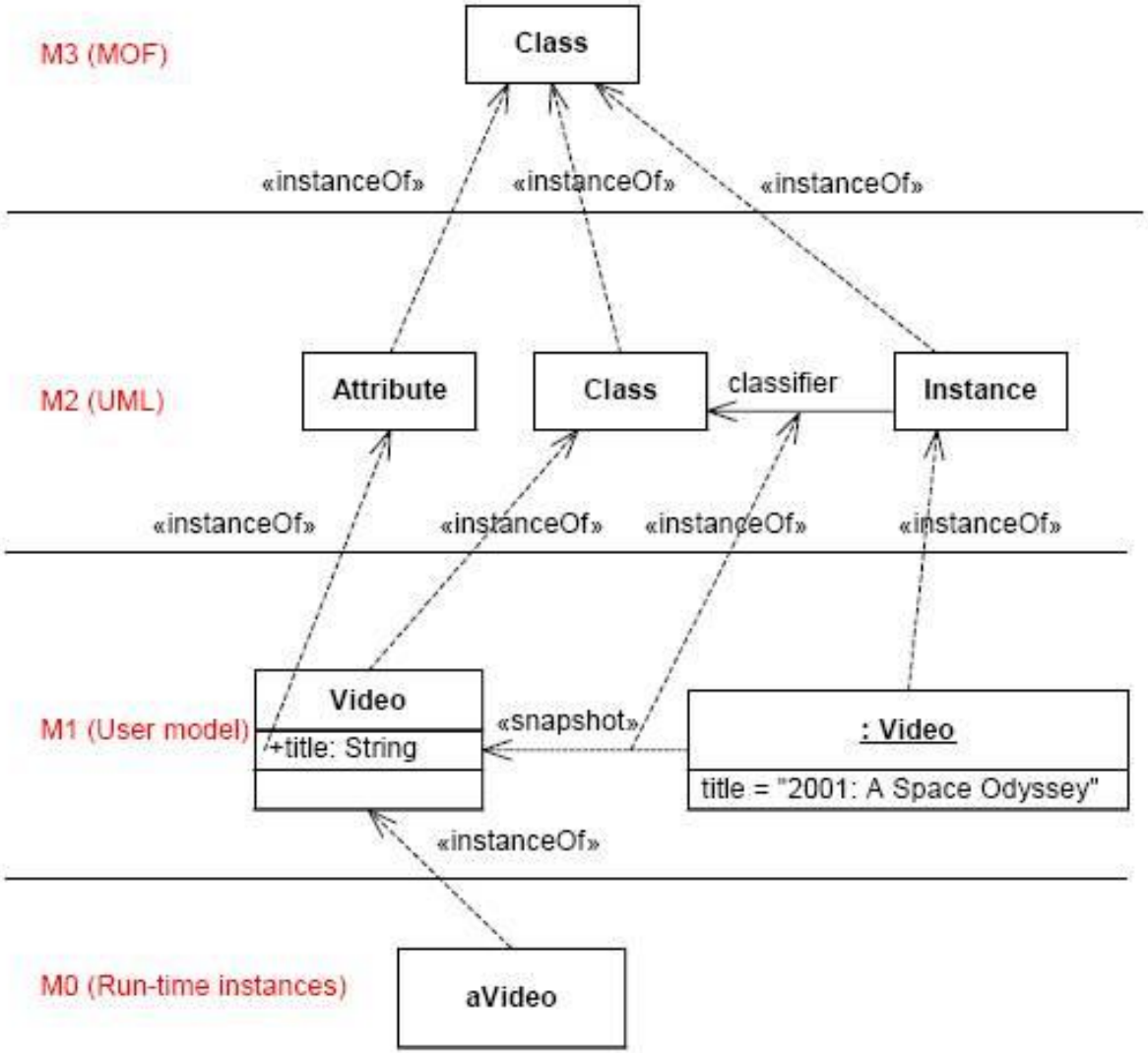
Per l'ingegnere del software quindi un modello costituisce una ***visione semplificata di un sistema*** che rende il sistema stesso più accessibile alla comprensione e alla valutazione e facilita il trasferimento di informazione e la collaborazione tra persone, soprattutto quando è espresso in forma visuale.

La produzione esplicita di modelli si rivela utile per le *comunicazioni tra i diversi attori di un processo di produzione* di software (committenti, analisti, progettisti, utenti, etc) che operano a diversi livelli di astrazione.

UML (2.0, OMG 2005)

- Il linguaggio UML fornisce uno standard notazionale per la rappresentazione dei modelli di un sistema software. Nasce nel 1997 dal lavoro congiunto di Booch, Rumbaugh e Jacoboson attingendo da concetti relativi alla metodologia di Booch, all'OMT di Rumbaugh e all'OOSE di Jacoboson.
- La sintassi e la semantica non viene descritta usando le convenzionali notazioni (BNF, EBNF, etc.) usate per i linguaggi testuali. Il meccanismo usato per descrivere UML prende il nome di metamodellazione
- UML 2.0 è definito da un ***metamodello*** e descrive di fatto un linguaggio, spesso limitandosi alla sintassi astratta e alla semantica.

Modelli e metamodelli



UML2: concetti-base (1)

Abstract class	A class that does not have objects instantiated from it
Abstraction	The identification of the essential characteristics of an item
Aggregation	Represents "is part of" or "contains" relationships between two classes or components
Aggregation hierarchy	A set of classes that are related through aggregation
Association	Objects are related (associated) to other objects
Attribute	Something that a class knows (data/information)
Class	A software abstraction of similar objects, a template from which objects are created
Cohesion	The degree of relatedness of an encapsulated unit (such as a component or a class)
Collaboration	Classes work together (collaborate) to fulfill their responsibilities
Composition	A strong form of aggregation in which the "whole" is completely responsible for its parts and each "part" object is only associated to the one "whole" object
Concrete class	A class that has objects instantiated from it
Coupling	The degree of dependence between two items

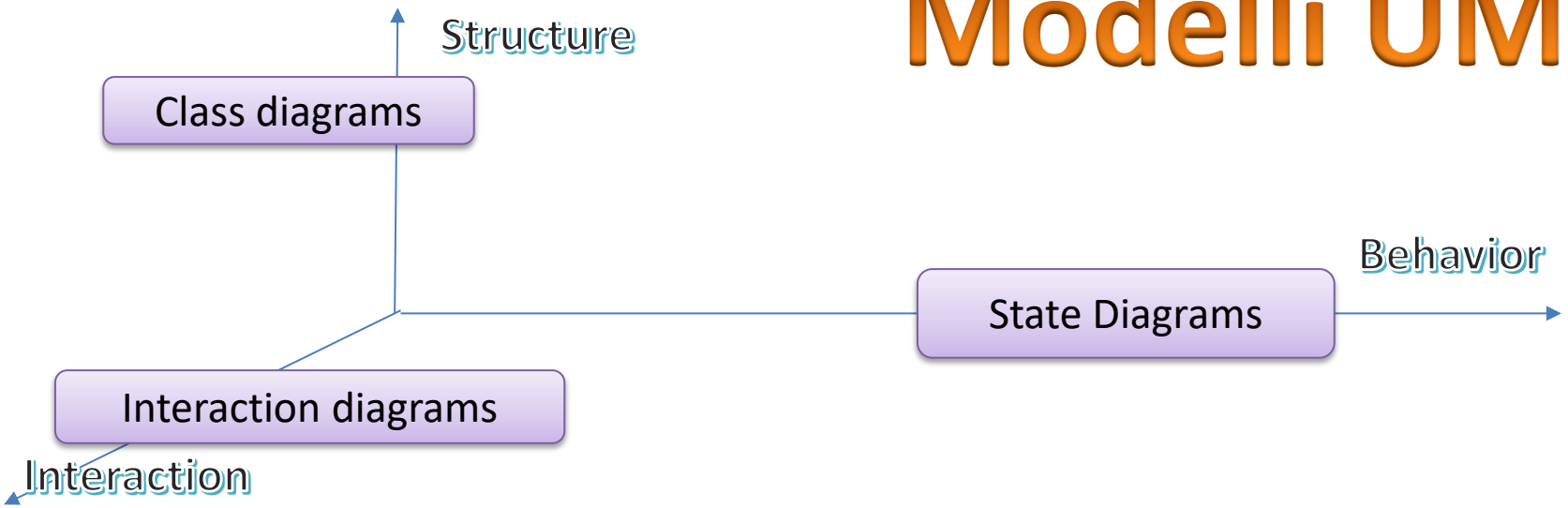
UML2: concetti-base (2)

Encapsulation	The grouping of related concepts into one item, such as a class or component
Information hiding	The restriction of external access to attributes
Inheritance	Represents "is a", "is like", and "is kind of" relationships.
Inheritance hierarchy	A set of classes that are related through inheritance
Instance	An object is an instance of a class
Instantiate	We instantiate (create) objects from classes
Interface	The definition of a collection of one or more operation signatures that defines a cohesive set of behaviors
Message	A message is either a request for information or a request to perform an action
Messaging	In order to collaborate, classes send messages to each other
Multiple inheritance	When a class directly inherits from more than one class
Multiplicity	A UML concept combining the data modeling concepts of cardinality (how many) and optionality.

UML2: concetti-base (3)

Object	A person, place, thing, event, concept, screen, or report
Object space	Main memory + all available storage space on the network, including persistent storage such as a relational database
Operation	Something a class does
Override	(redefine) attributes and/or methods in subclasses
Pattern	A reusable solution to a common problem taking relevant forces into account
Persistence	The issue of how objects are permanently stored
Persistent object	An object that is saved to permanent storage
Polymorphism	Different objects can respond to the same message in different ways, enable objects to interact with one another without knowing their exact type
Single inheritance	When a class directly inherits from only one class
Stereotype	Denotes a common usage of a modeling element
Subclass	If class "B" inherits from class "A," we say that "B" is a subclass of "A"
Superclass	If class "B" inherits from class "A," we say that "A" is a superclass of "B"
Transient object	An object that is not saved to permanent storage

Modelli UML



modelli dello stato (*vista statica*): descrivono le strutture statiche dei dati, e si possono ottenere utilizzando per esempio i **diagrammi delle classi**;
modelli del comportamento (*vista operativa*): descrivono la collaborazione tra oggetti. Ci sono molte tecniche di visualizzazione per la modellazione del comportamento, come il ***diagramma dei casi d'uso, il diagramma di sequenza, il diagramma di collaborazione e il diagramma di attività***;
modelli del cambiamento di stato (*vista dinamica*): descrivono gli stati permessi dal sistema nel tempo. La prima tecnica di visualizzazione è il **diagramma degli stati**, basato su un modello di evoluzione degli stati di un oggetto.

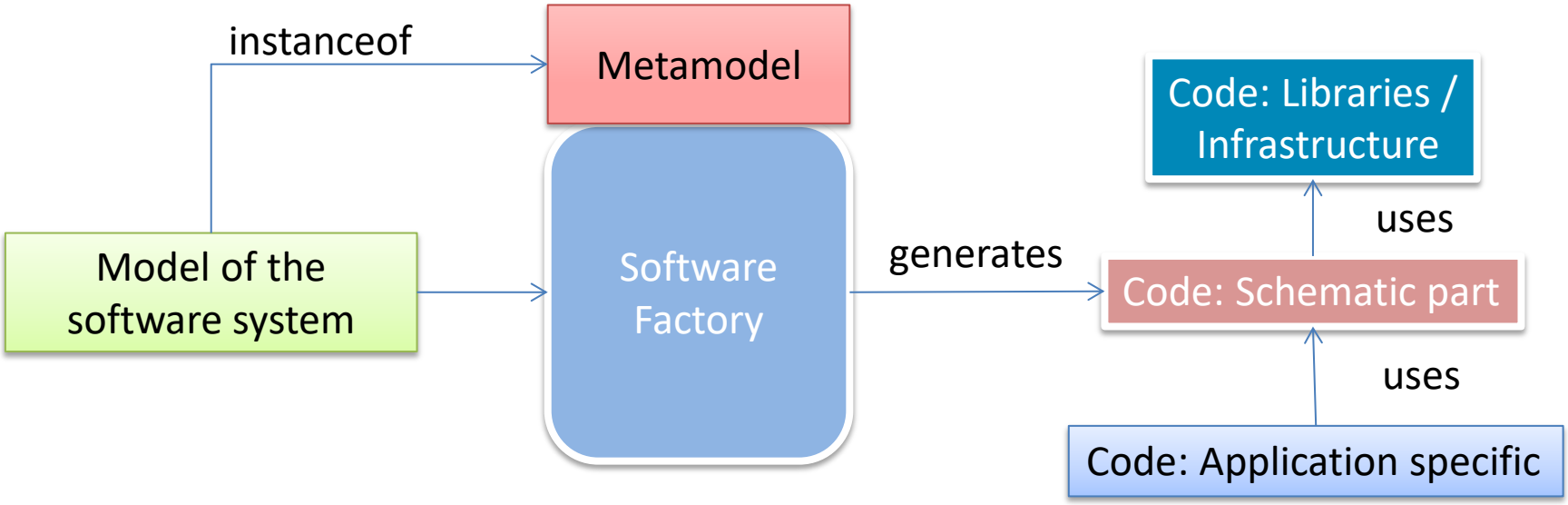
UML2 messaging

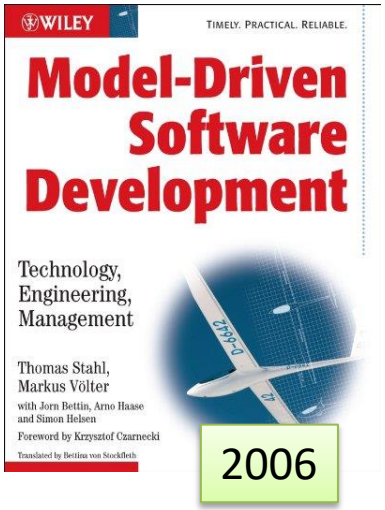
- A *message* defines a particular communication between lifelines of an interaction.
- Objects respond to messages *through the invocation of an operation* (*classes* do so through the invocation of static operations)
- *Abstracting away remote behavior is an anti-pattern* (A Note on Distributed Computing)

Il ruolo dei Modelli

- Fondamentale sarebbe la possibilità di pensare ai modelli come una ***nuova forma di codice sorgente*** e di disporre di generatori capaci di ricavare automaticamente codice eseguibile dai modelli.

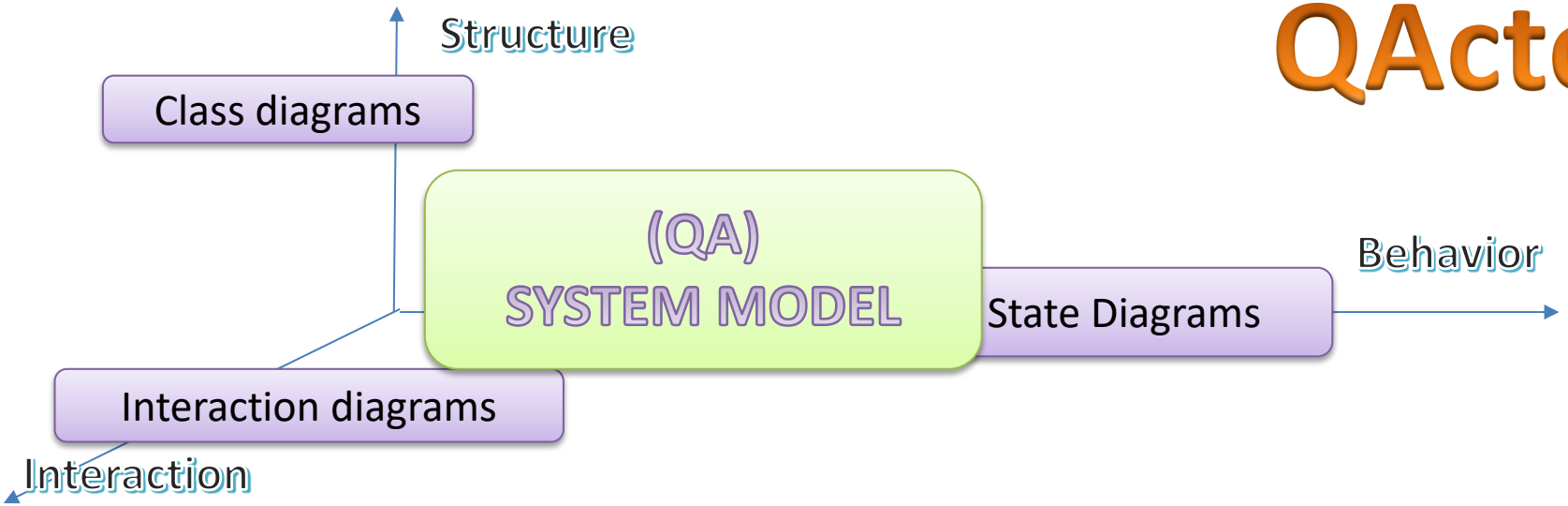
MDSD





- Model-Driven Software Development (MDSD) puts **analysis and design models** on par with code.
- Models do not constitute documentation, but are considered equal to code, as their implementation is automated.
- The goal of the book is to convince you, the reader, that MDSD is a practicable method today, and that it is superior to conventional development methods in many cases.

QActors



Main Concepts

QActors : *active* entities that:

- *behave* in a **context** as a **finite state machine**
- *interact* by means of **messages**
- can handle **events**
- can perform built-in or user-defined (implemented in objects) **actions** that:
  - realize algorithms (*terminate*)
  - can **react** to events

CaseStudy1: qa model

```
System radargui
Event polar              :  p( Distance, Angle  )        //emitted by the plant (on Raspberry)
Dispatch polarMsg    :  p( Distance, Angle  )        //forwarded by the tester

Context  ctxRadarBase    ip [ host="localhost"  port=8033 ]
QActor radarguibase context ctxRadarBase  {
Plan init normal [
            actorOp activateGui
 ]
switchTo waitSonarInfo
Plan waitSonarInfo [ actorOp noOp]
     transition stopAfter 86400000  //one day
     whenEvent polar      ->   handleSonarInfo ,
     whenMsg  polarMsg -> handleSonarInfo
     finally repeatPlan
Plan handleSonarInfo resumeLastPlan [
            onMsg  polarMsg : p(D,A)  ->  actorOp sendDataToGui( D,A )  ;
            onEvent polar       : p(D,A)   -> actorOp sendDataToGui( D,A )
 ]
}
```

About Testing

White box testing:	testing with knowledge of the target source code
Black box testing:	testing on the target public API without knowledge of the target source code
Unit Testing:	testing single units of work (white box)
Integration Testing:	testing how different units of work interact
Functional Testing:	testing subsystems (usually on a boundary API)
Stress/Load Testing:	testing the system performance
(User) Acceptance Testing:	testing the system as a user

Test Coverage;

white box testing : percentuale di righe di codice applicativo eseguite dalla suite di test;

black box testing: numero di metodi eseguiti sulle unità testate

failure la situazione in cui il software opera in contrasto con le specifiche

fault l'elemento del software che causa una failure.

About Testing

Un functional testing potrebbe dare risultato positivo anche se lo stato interno del sistema è erroneo. D'altra parte un piano di collaudo basato su testing strutturale può distogliere l'attenzione su cosa investigare. (il ***code coverage*** non sempre garantisce di raggiungere ***functional coverage***).
La ***structural coverage*** può essere usata come una misura di adeguatezza di collaudi funzionali.

Una possibile metodologia {Mtrick94]:

- I test funzionali sono generati dai requisiti o dalle specifiche e dal progetto, con l'intento di individuare le situazioni di fallimento.
- La copertura strutturale è esaminata solo dopo che i test funzionali sono stati tutti soddisfatti.
- Nel caso la copertura strutturale sia da completare, il collaudatore non si lascia guidare dal sistema software, ma cerca di generare nuove situazioni a livello funzionale.

CaseStudy1: testing

```
QActor qatester  context ctxRadarBase {
Rules{
p(80,0).
p(80,30).
p(30,50).
p(80,60).
p(20,70).
p(80,90).
p(80,120).
p(10,130).
p(80,150).
p(80,180).
}
Plan init normal [
  delay  1000 ;
     [ !? p(X,Y) ] forward radarguibase -m polarMsg : p(X,Y) else endPlan "testDone" ;
     [ ?? p(X,Y) ] emit polar : p(X,Y) else endPlan "testDone"
 ]
 finally repeatPlan
}
```