Component-based Robotics Engineering

Davide Brugali, Luca Gherardi
Università degli Studi di Bergamo
Anchorage, AK, USA
May 7, 2010

IEEE Tutorial

- Component-based Robotics Engineering
  - Part I: Reusable building block
    - IEEE RAM, December 2009
  - Part II: Systems and Models
    - IEEE RAM, March 2010

Introduction

- TODAY: A lot of robotics software are available but often not reusable
  - They are tight to specific robot
- GOAL: build robotic software through a composition of reusable building blocks
  - called component
- Solution: Component Based Software engineering

What is reuse?

- Software reuse is:
  - the practice of developing software,
  - from a stock of building blocks,
  - so that similarities in requirements and/or architecture between applications ca be exploited,
  - to achieve substantial benefits in productivity, quality and business performance.

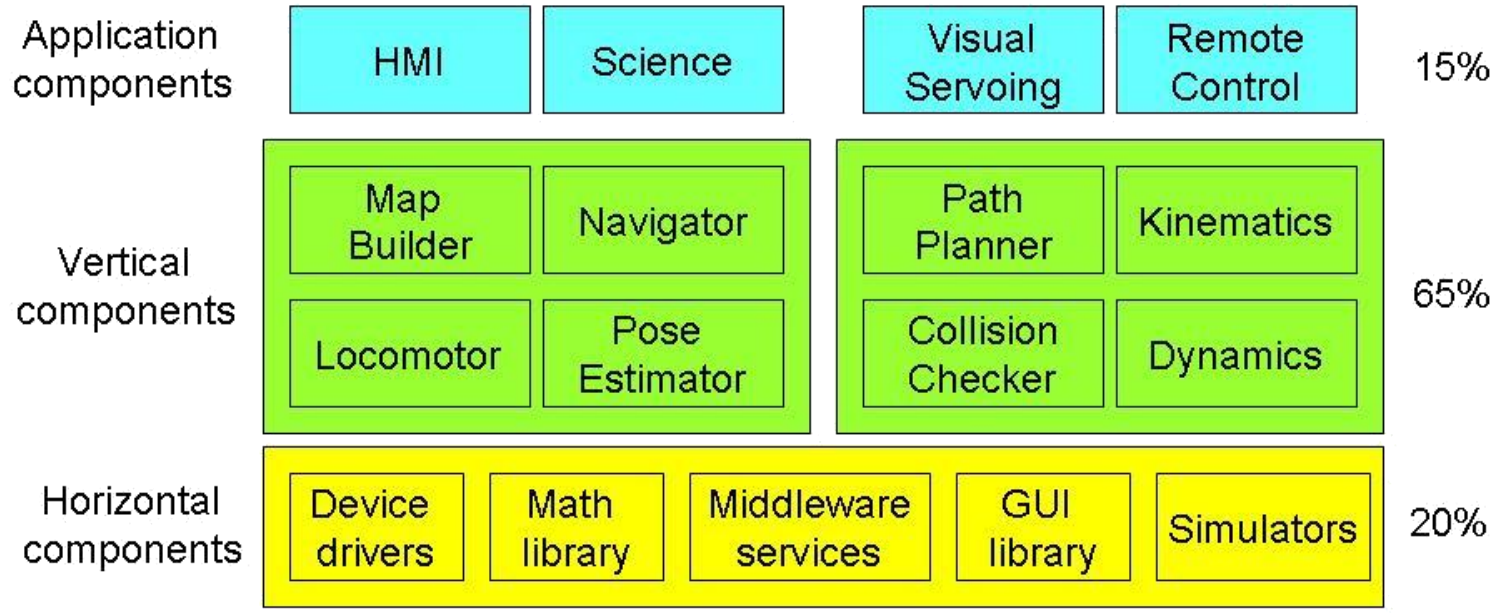
What makes a component reusable

- Three aspects are equally important

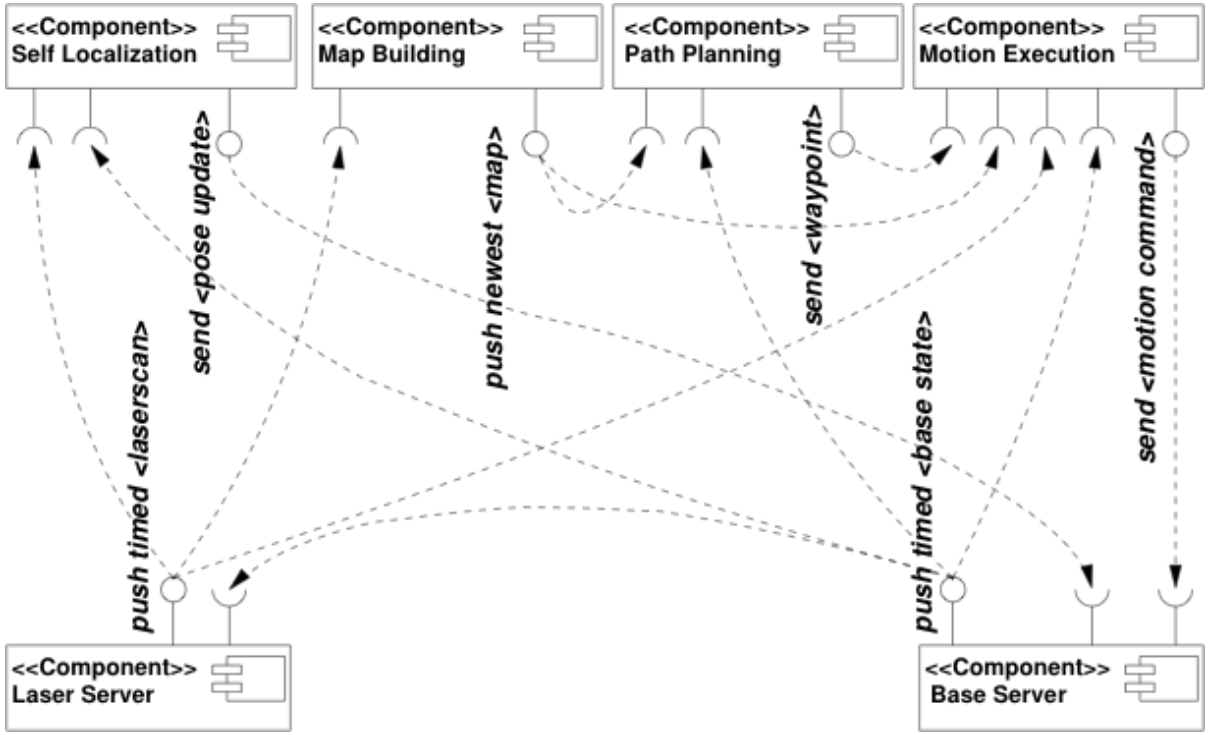
  - **Quality.** The component has to be usable (reliability, performance, efficiency,…),
  - **Functionality.** It should be desirable to reuse the component (it offer a function that is needed),
  - **Technique.** It should be possible to reuse the component (portability, interoperability, modularity).

Types of reuse

- Horizontal component: provide functionality that can used in totally different use cases.
- Vertical component: provide functionality that can be used in a specific domain.

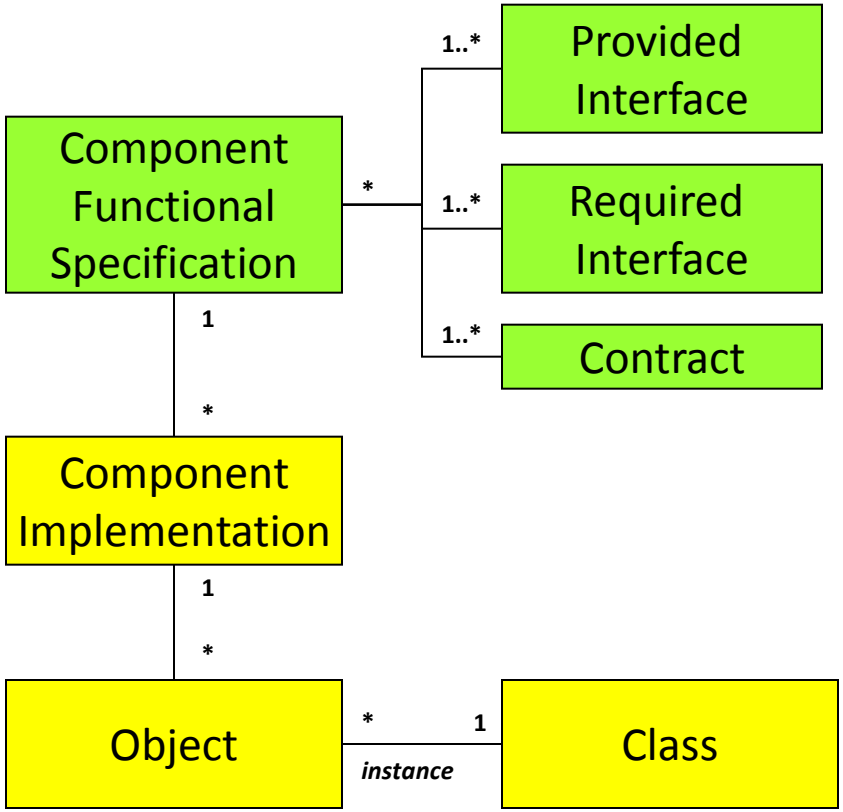


Component-Based System



- Component: a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third part.

Component

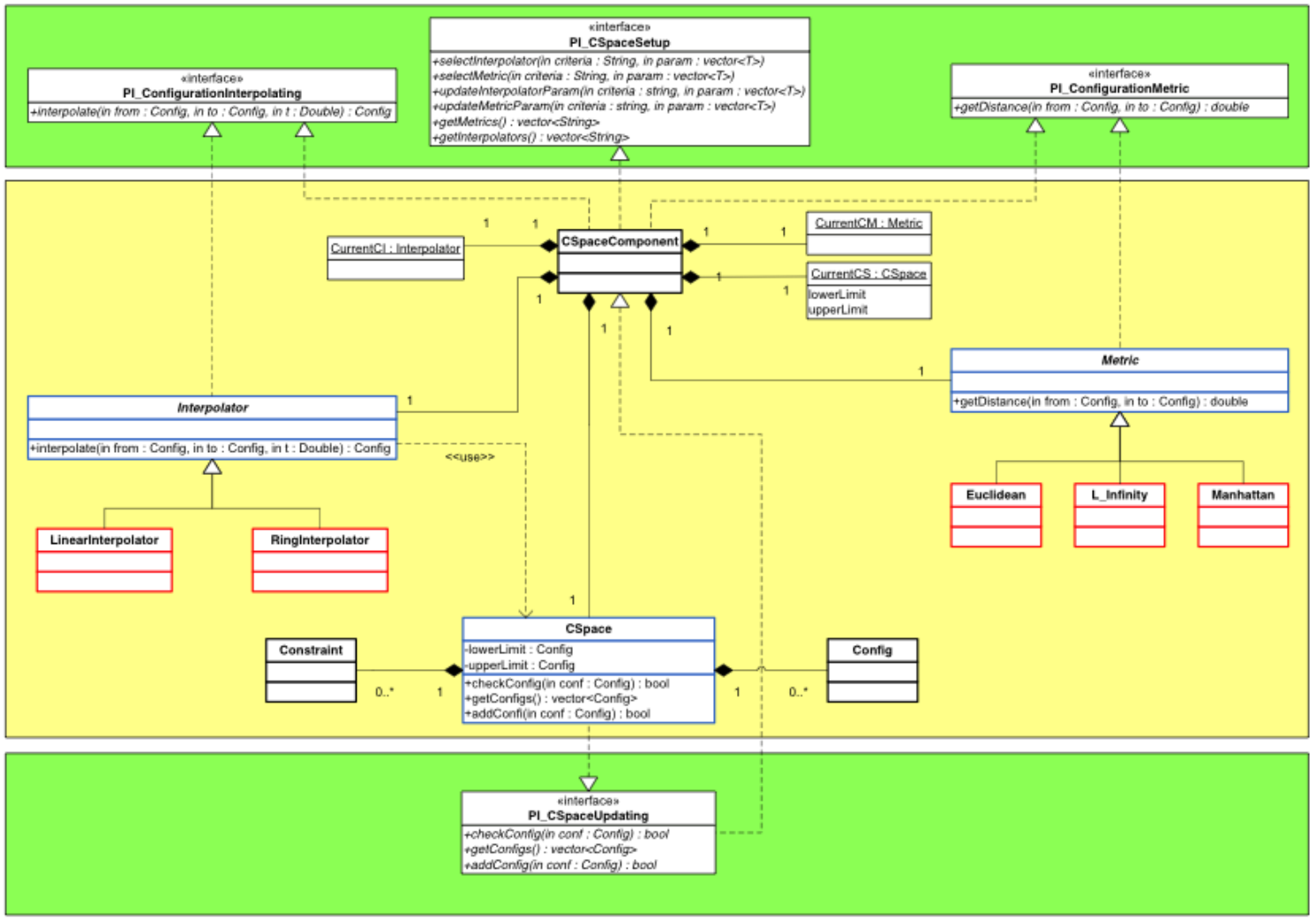


- Separation of specification and implementation

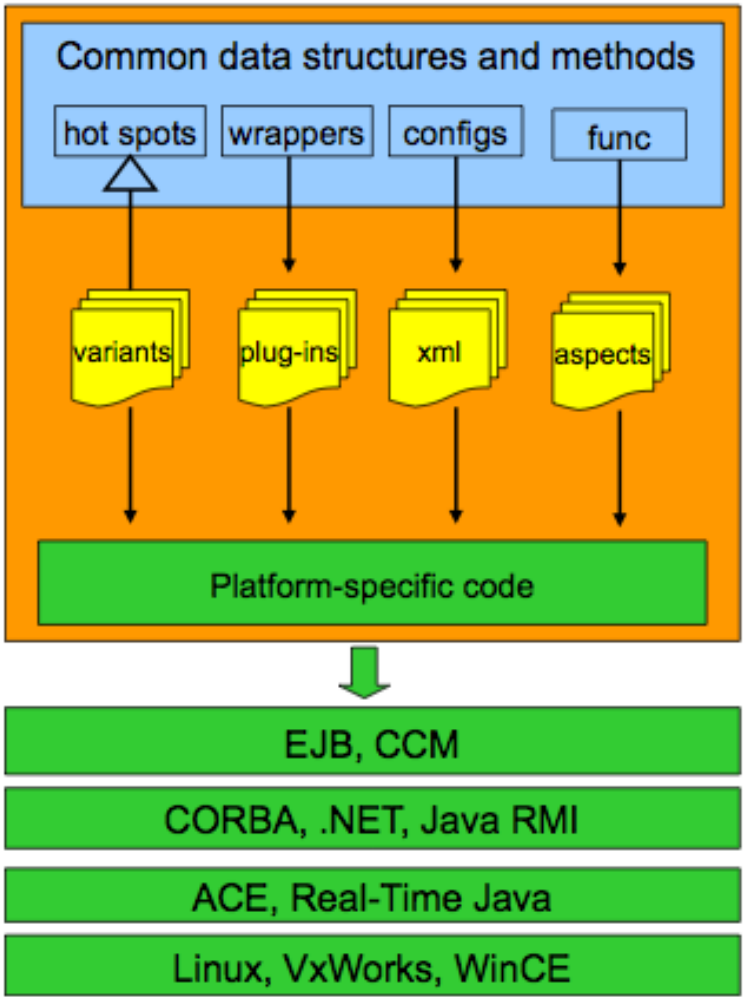
Component Specification

- Interface design concept
  - Provided/Required
  - Service/Data
  - Strongly-typed/Loosely-typed
  - Stateful/Stateless
  - Minimal/Complete
- Contract

Implementation example



Component framework



- A skeleton of a component implementation that can be specialized by a component developer to produce custom component.

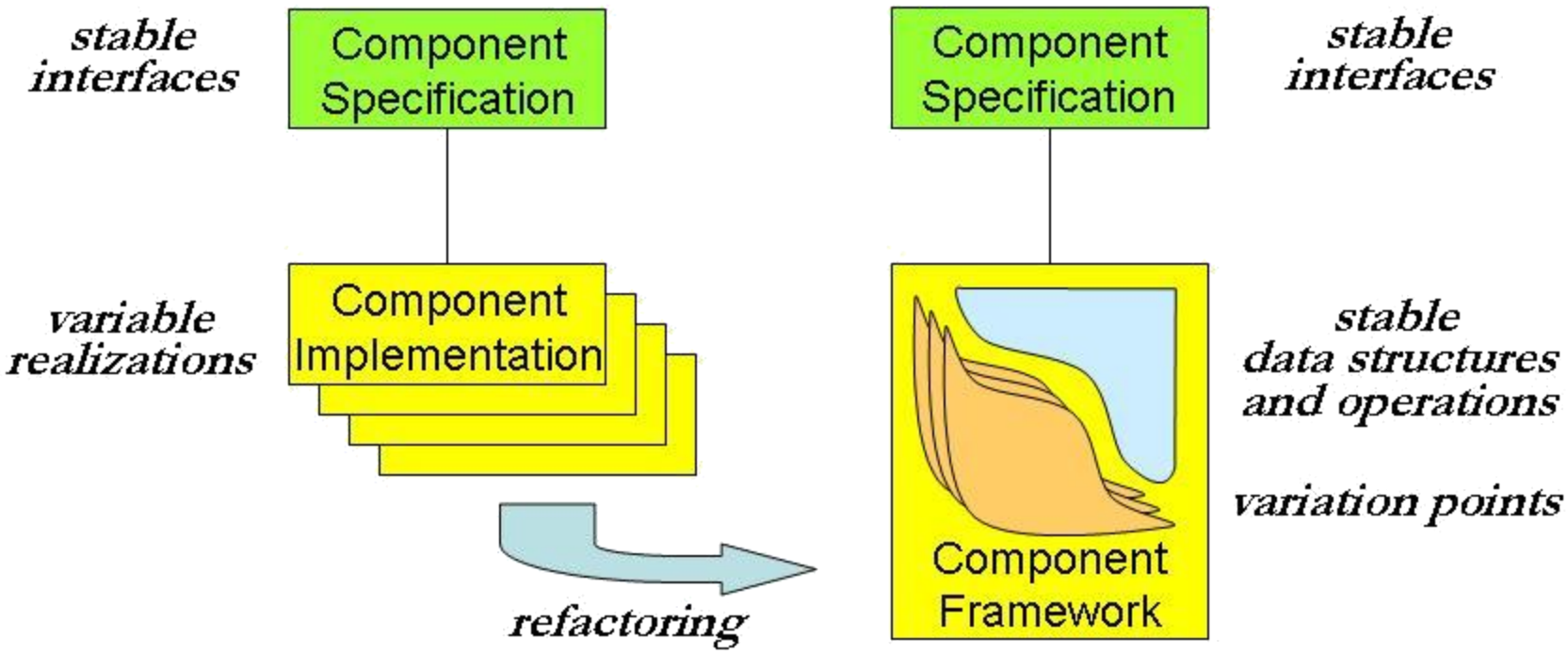
- Stable point
- Variation point
- Variant

- Life span

Implementing variability

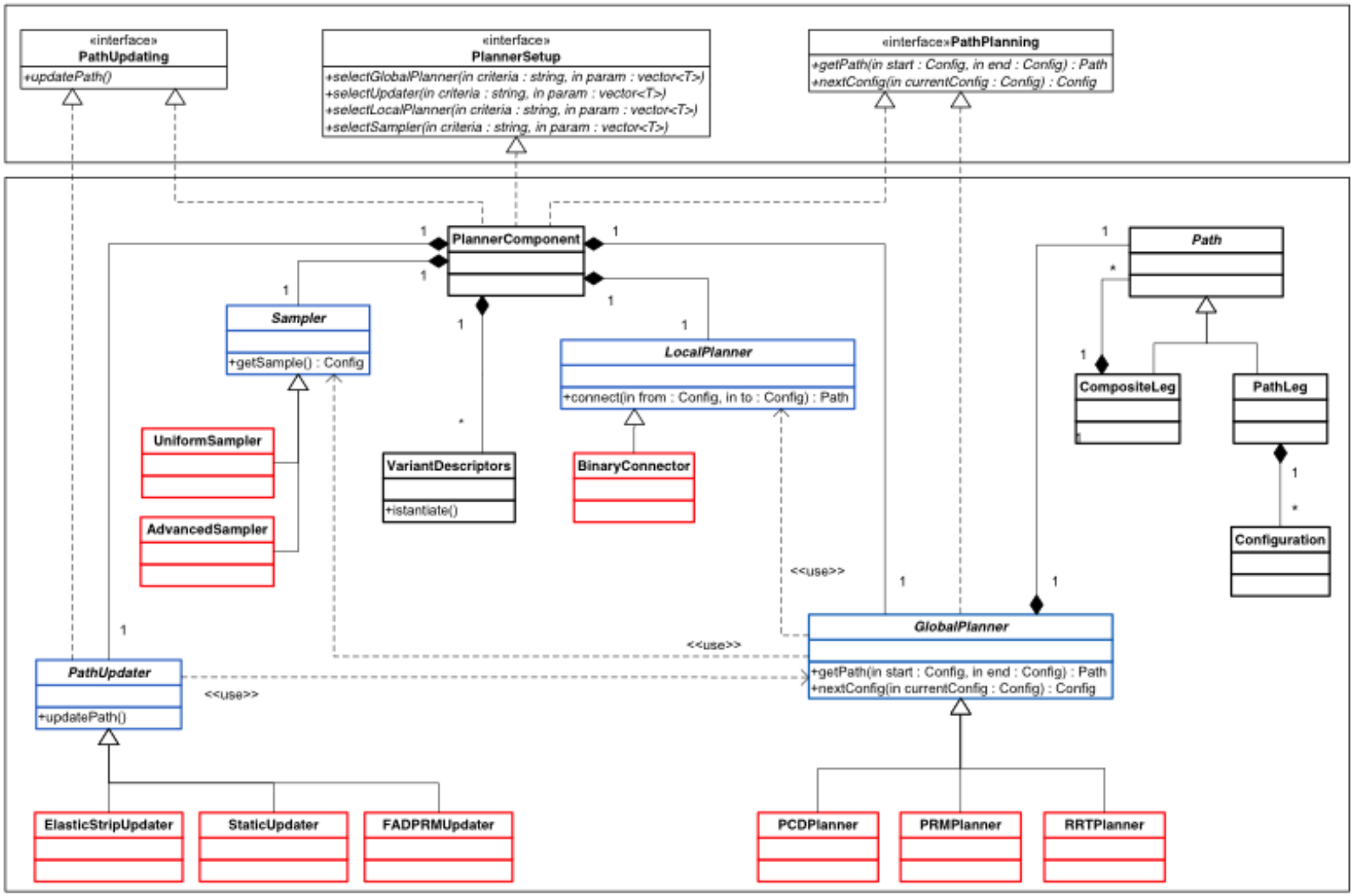
- Classification according to binding time
  - Compile time, Link time, Run time
- Technologies
  - Inheritance and extension
  - Aggregation and delegation
  - Parameterization
  - Conditional compilation
  - Dynamic Link Libraries
  - Reflection

Refactoring

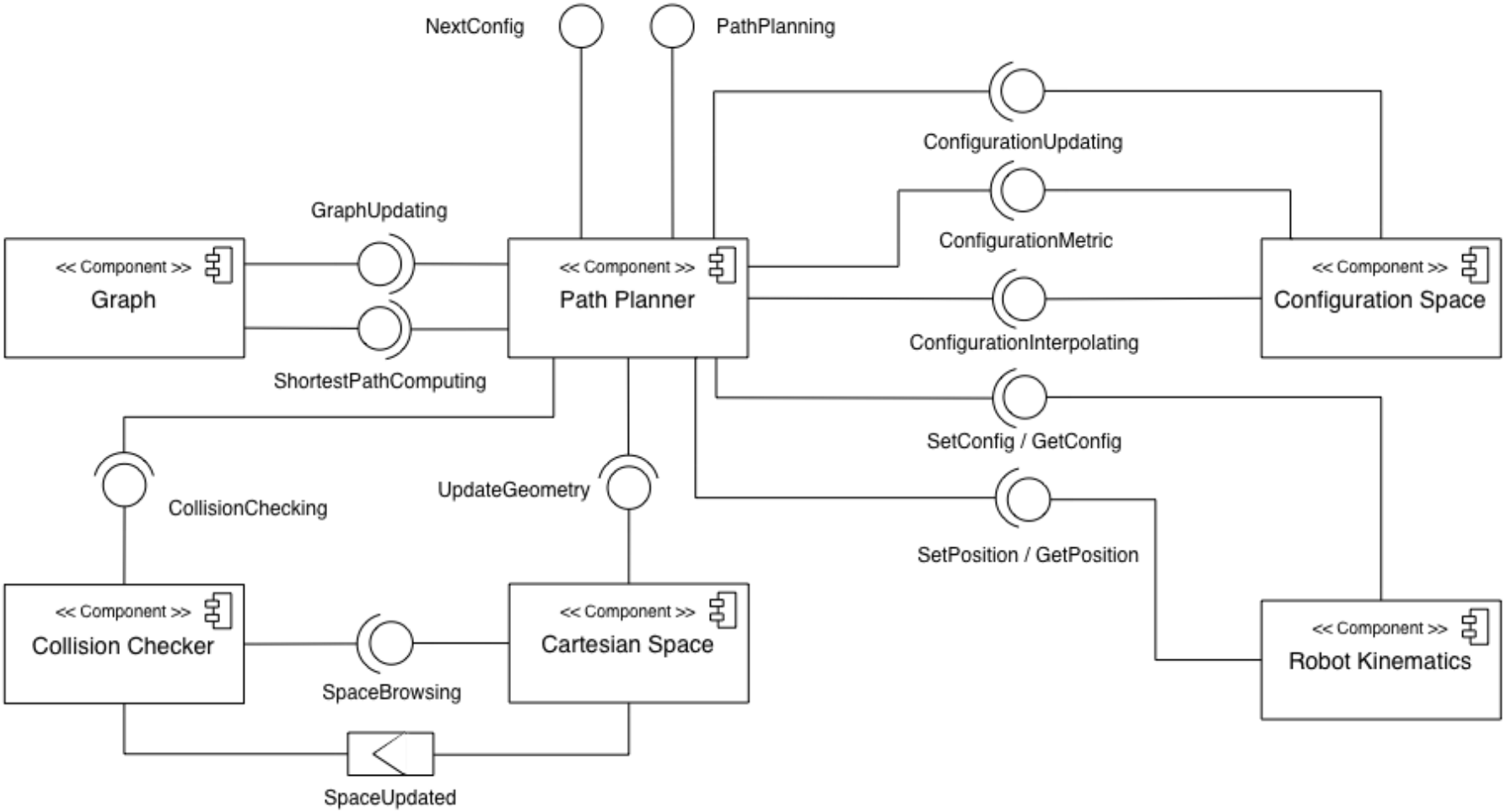


- A technique that aims to restructure a set of existing software libraries without affecting their external behavior in order to harmonize their architecture, data structures, and APIs.

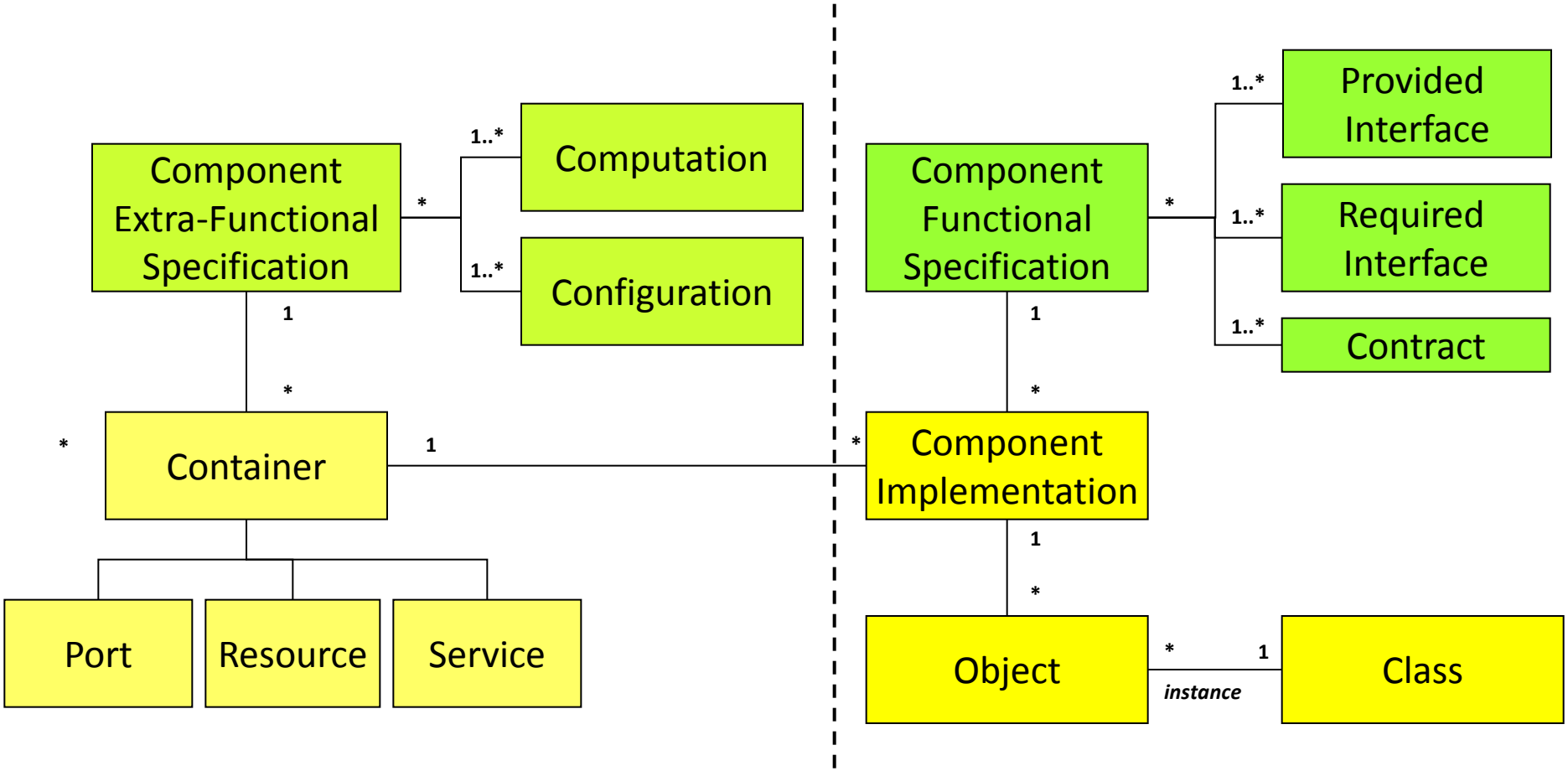
Path planner component-framework



Motion planning
Component-based system



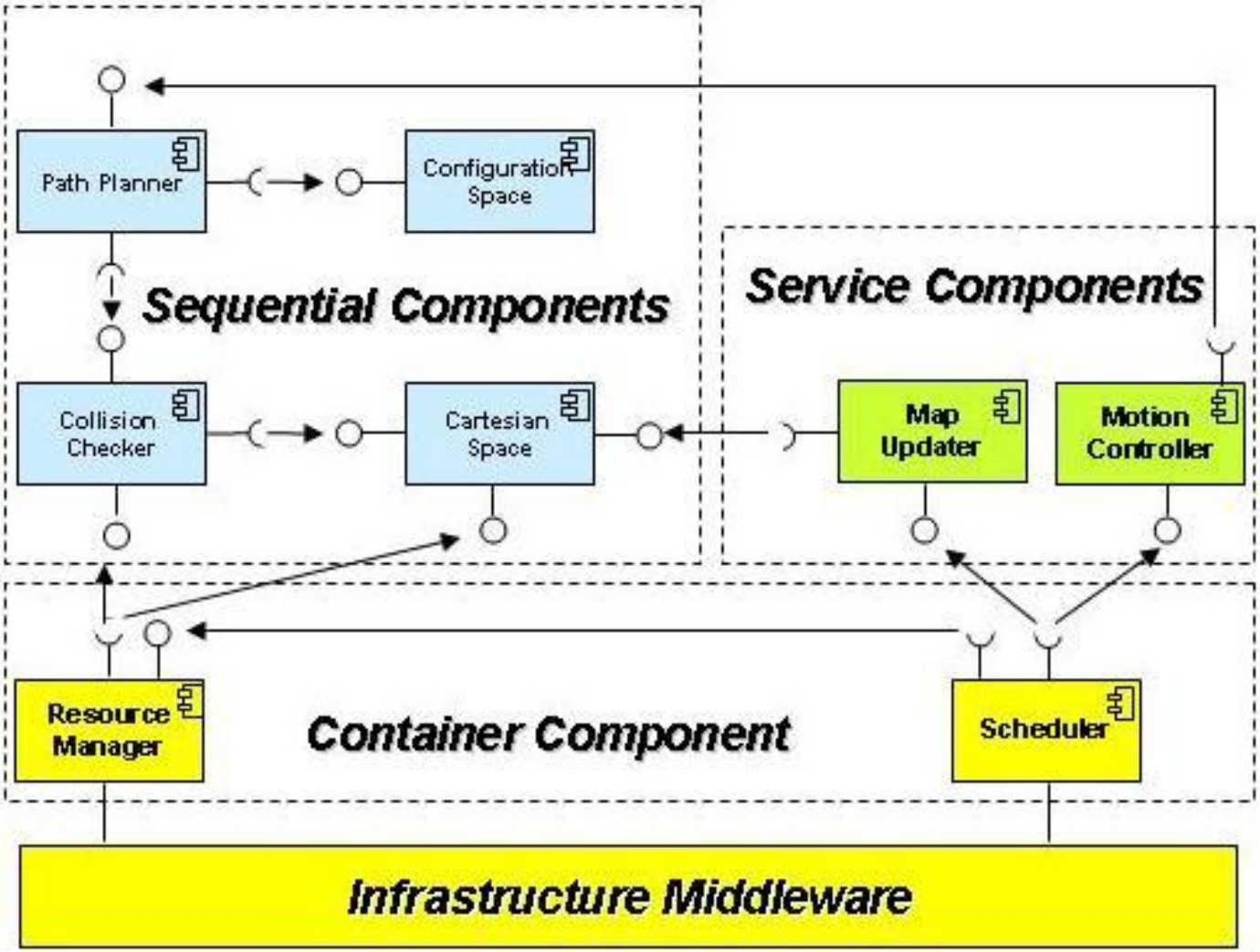
Separation of functional and extra-functional requirements



Computation

- Is concerned with the data processing algorithms
  - Data transformation -> data flow design
  - Control transformation -> control flow design
- Different levels of concurrency's granularity
  - Fine grain
  - Medium grain
  - Large grain

Granularity of control transformation



Components Assembly

n sequential Components

+

m service Components

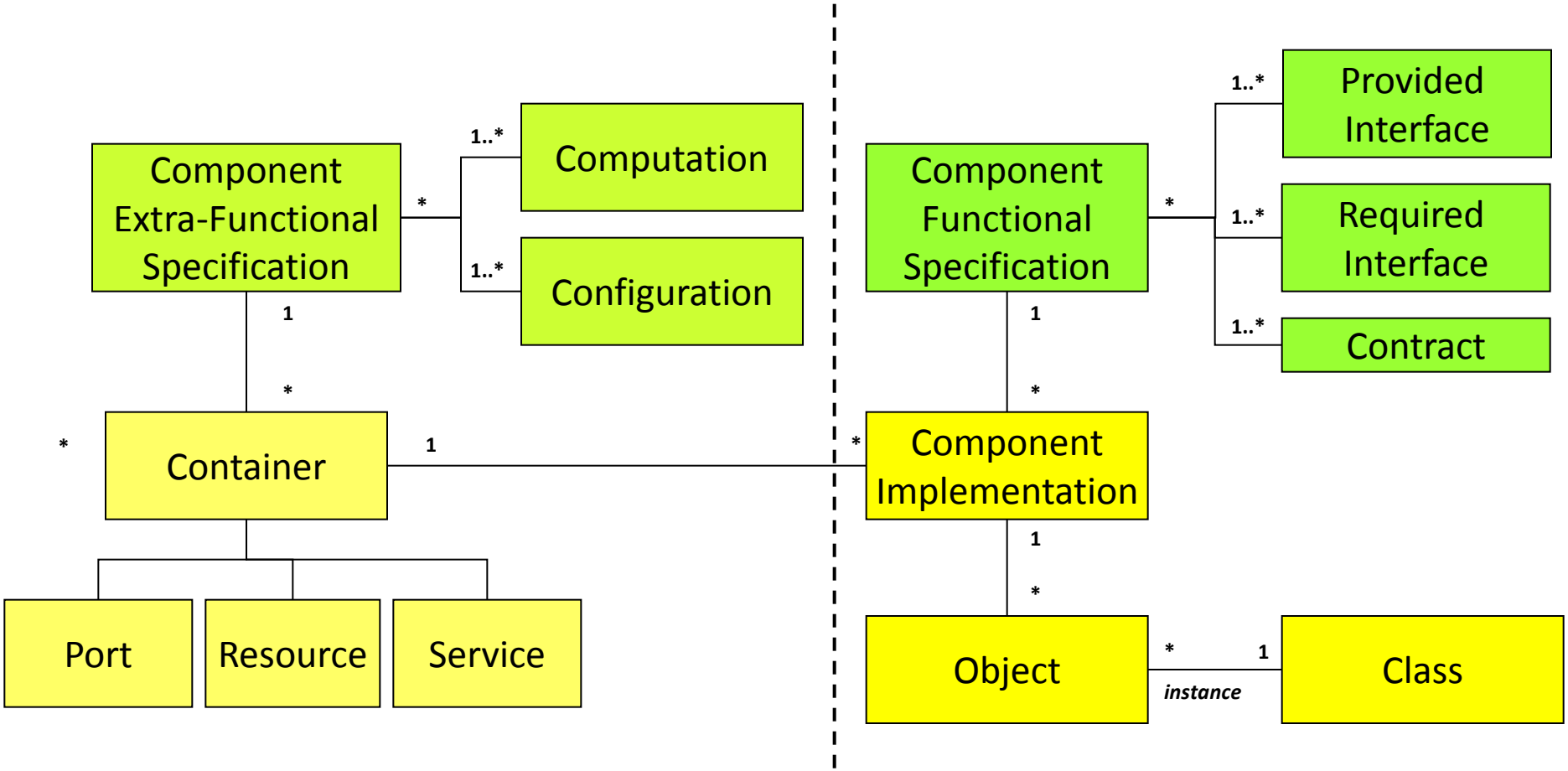
+

1 container Component

=

Component Assembly

Separation of functional and extra-functional requirements



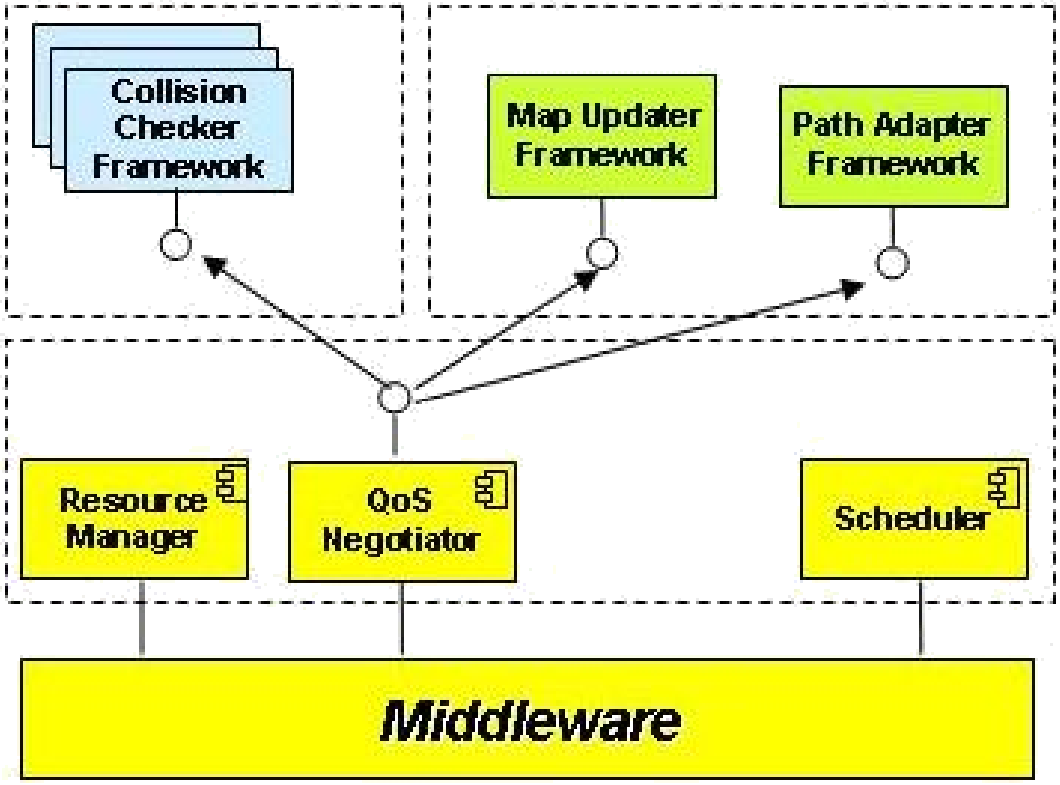
Configuration

- Determines which system components should exist, and how they are inter-connected

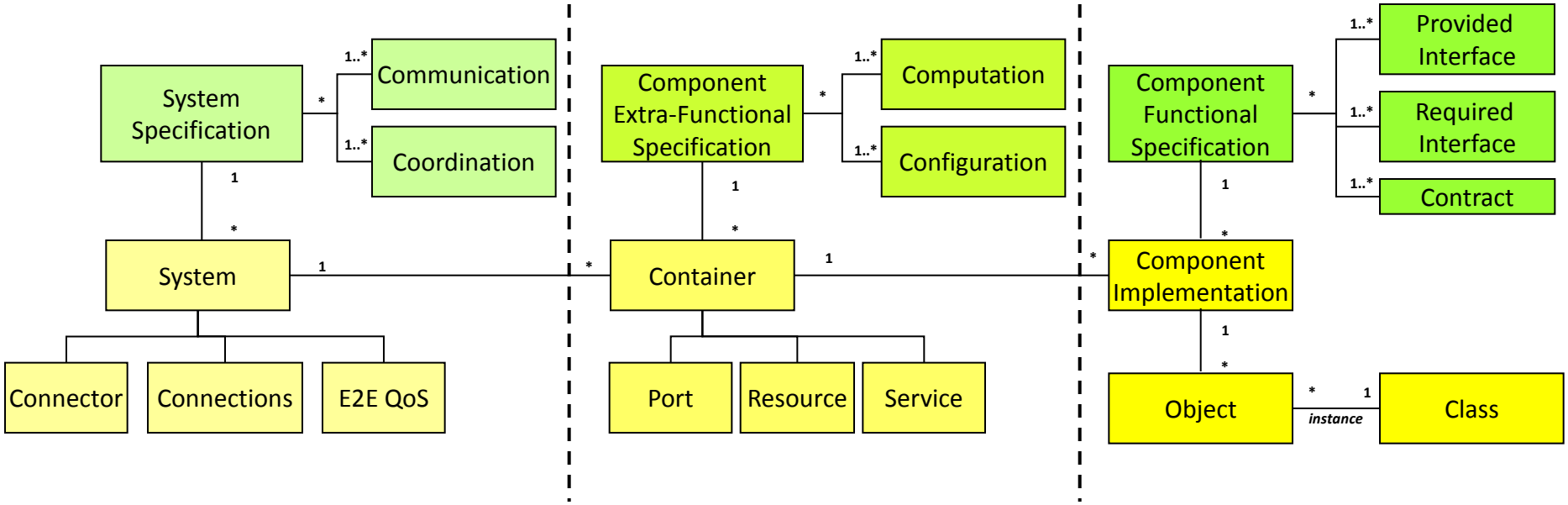
- A configuration is described in term of
  - Components
  - Connection between component
  - Connectors: architectural building blocks used to model interactions among components and rules that govern those interactions

Quality of Service

- QoS depends on implementation, availability of resources and environment



Separation of component and system requirements



Communication

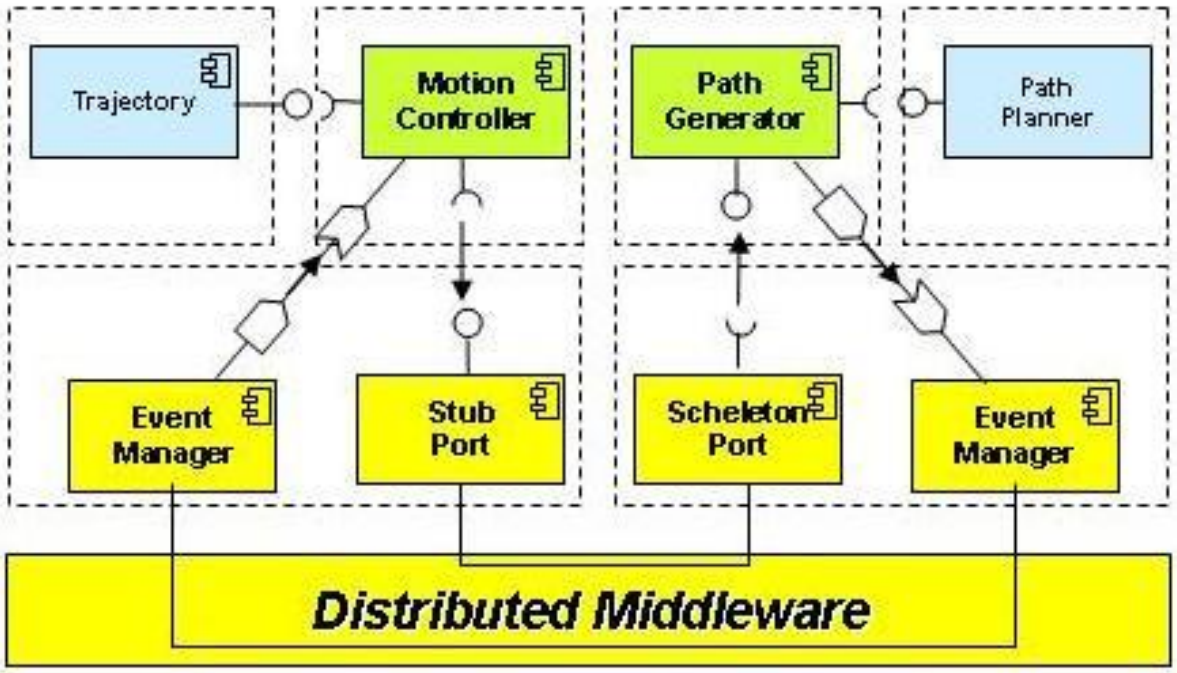
- Deal with the exchange of data
- Two types of communication
  - Imperative: caller/provider mechanism
  - Reactive: broadcaster/listener mechanism
- Visibility implies dependencies and influence reusability of components.
- Three dimensions of decoupling
  - Space / Time / Synchronization decoupling

Decoupling degree between component

- Three dimensions of decoupling
  - Space decoupling
  - Time decoupling
  - Synchronization decoupling

- Communication paradigm influence the decoupling degree

Communication paradigm



- Communication paradigm influence the decoupling degree
  - Remote method invocation
  - Publish/Subscribe

Coordination

- Is concerned with the interaction of the various system components

- Coordination language & models
  - Data-driven
  - Control-driven

Interaction between components