

## Software Engineering Principles for Complex Systems

TDA594

# Modularity-2

Sam Jobara  
[jobara@chalmers.se](mailto:jobara@chalmers.se)  
Software Engineering Division  
Chalmers | GU



# Learning Objectives

- ◆ **Architecture styles for modularity**
  - ◊ Reduce SPLs complexity
- ◆ **Robotics Modular Architecture**
  - ◊ Modular autonomous Robots
- ◆ **Feature-oriented programming**
  - ◊ Mapping Modules with Features

## Main Reference:

Feature-Oriented Software Product Lines: Concepts and Implementation,  
Sven Apel • Don Batory • Christian Kästner • Gunter Saake

Other publications (see slides for references)



# Architectural Styles

Reduce SPLs complexity



# Architectural Styles

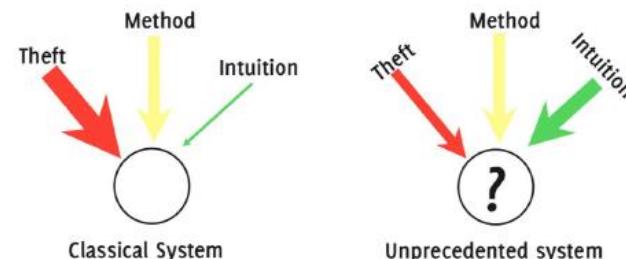
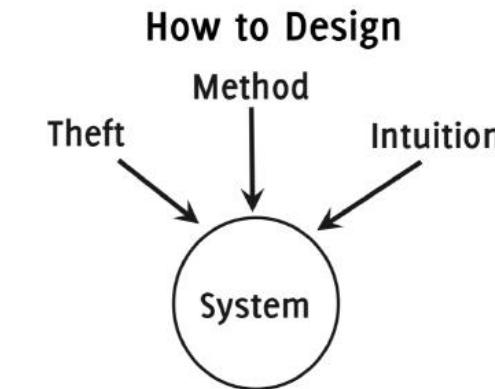
## Where do Architectures Come From?

Method -Systematic way

- Efficient in familiar terrain
- Scalable and predictable outcome
- Not always successful
- Quality of the methods varies

Creativity –Intuition

- Fun!
- May yield the best result
- May be dangerous
- May be unnecessary
- Theft –Reuse System of the same kind





# Architectural Styles

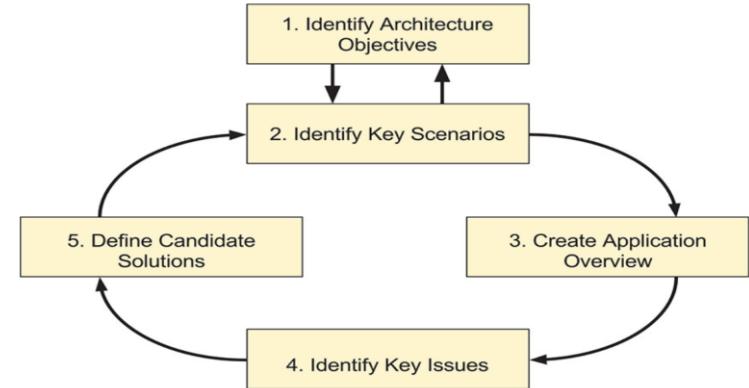
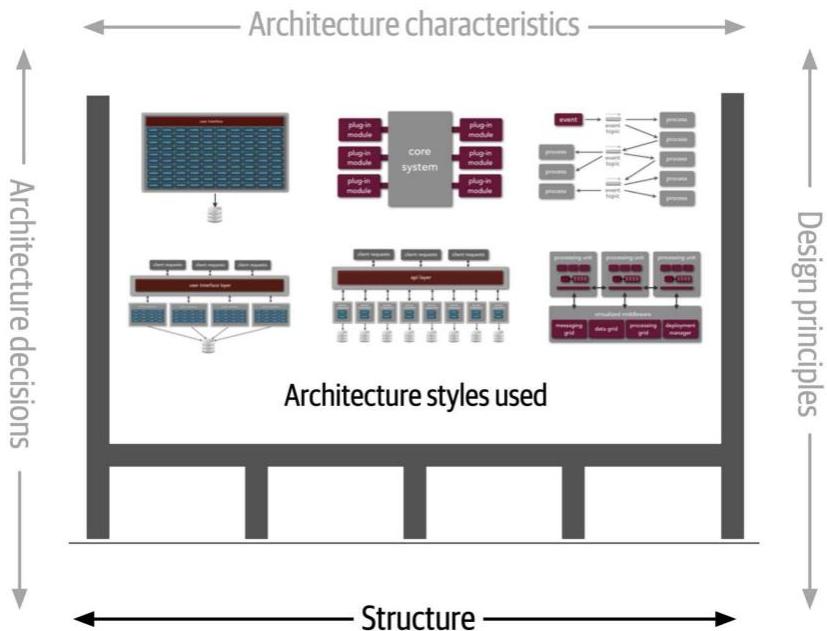


FIGURE 7.5 Iterative steps of the technique for architecture and design

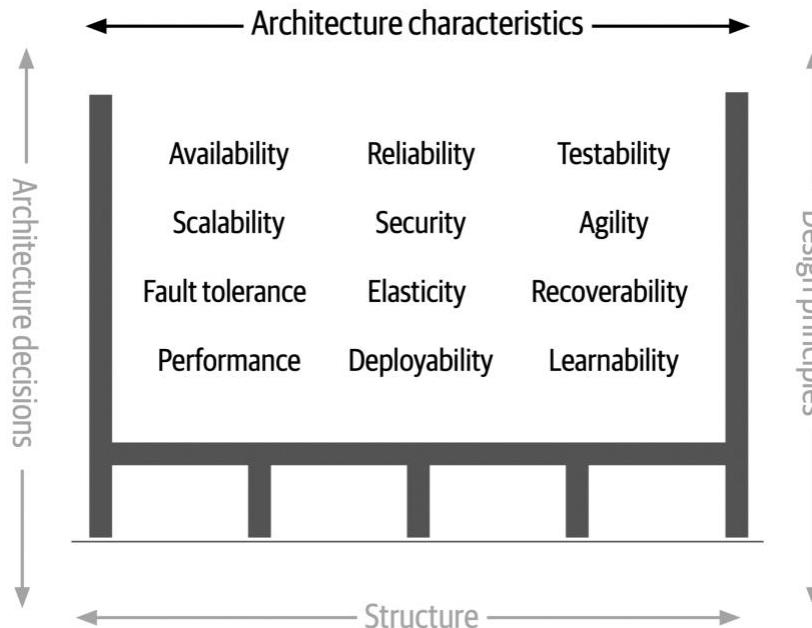
## Example:

- client-server
- P-to-P
- Microservices
- layered
- Event-driven



# Architectural Characteristics

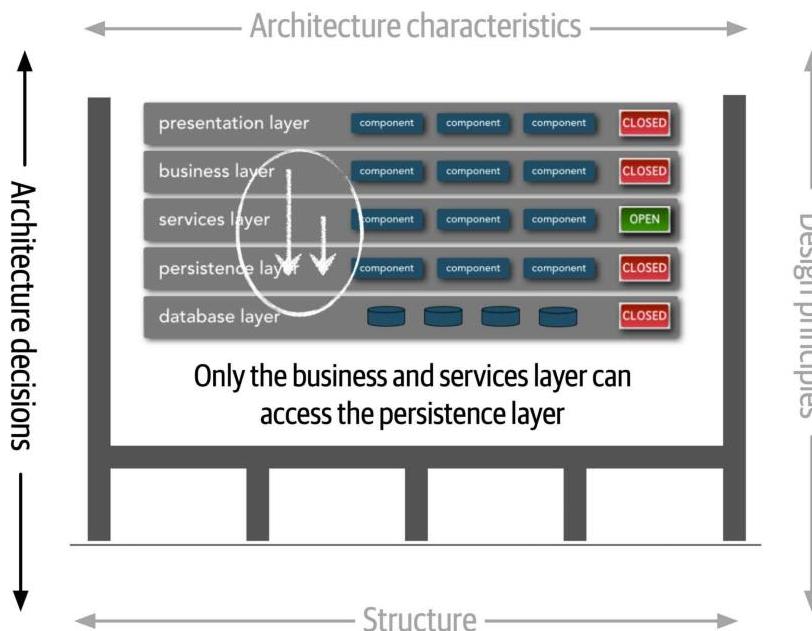
System quality attributes which encompasses all qualities





# Architectural Decisions

Define the rules for how a system should be constructed



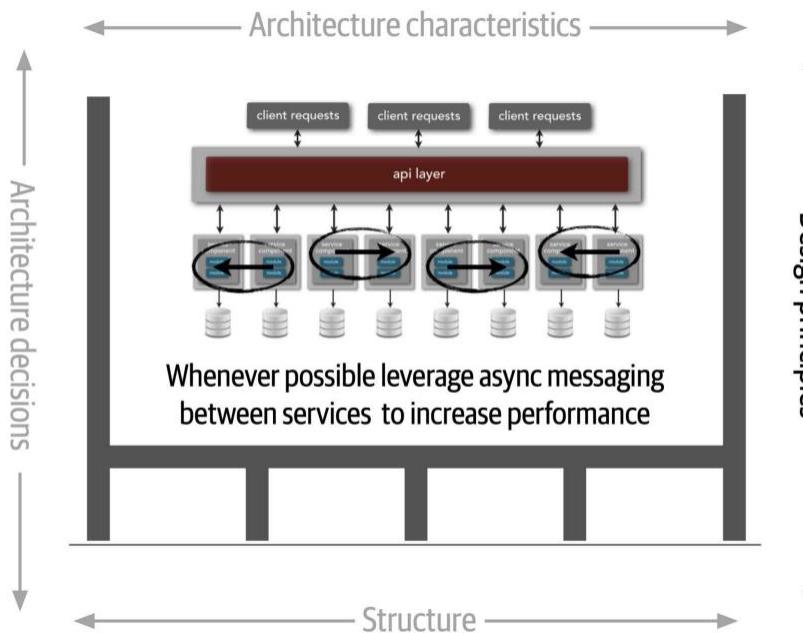
## Example:

- SoC
- Information hiding
- Abstraction
- Security controls
- Data access
- Synchronization



# Architectural Principles

A design principle is a guideline rather than a hard-and-fast rule



## Guidelines:

- Power consumption
- Documentations
- Test iterations & coverage



# Architectural Styles

## Microservices

Popular for modularity, and fast CD for DevOps, gained significant momentum in recent years due to mobile and cloud computing.

According to a [recent O'Reilly radar survey](#) on the growth of cloud computing, stated that 52 percent of the 1,283 responses say they use microservices concepts, tools, or methods for software development.

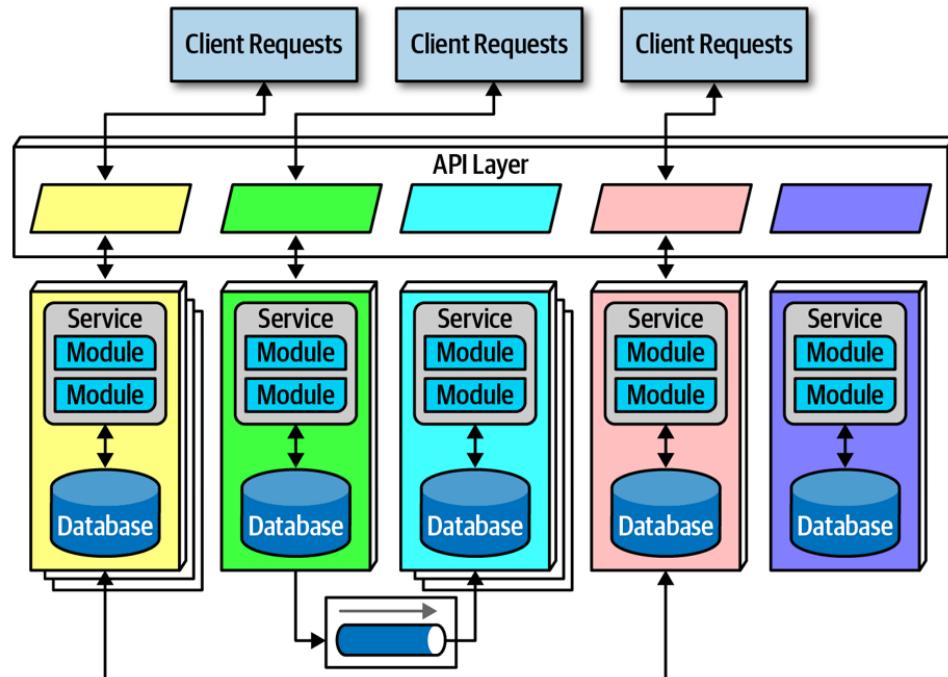


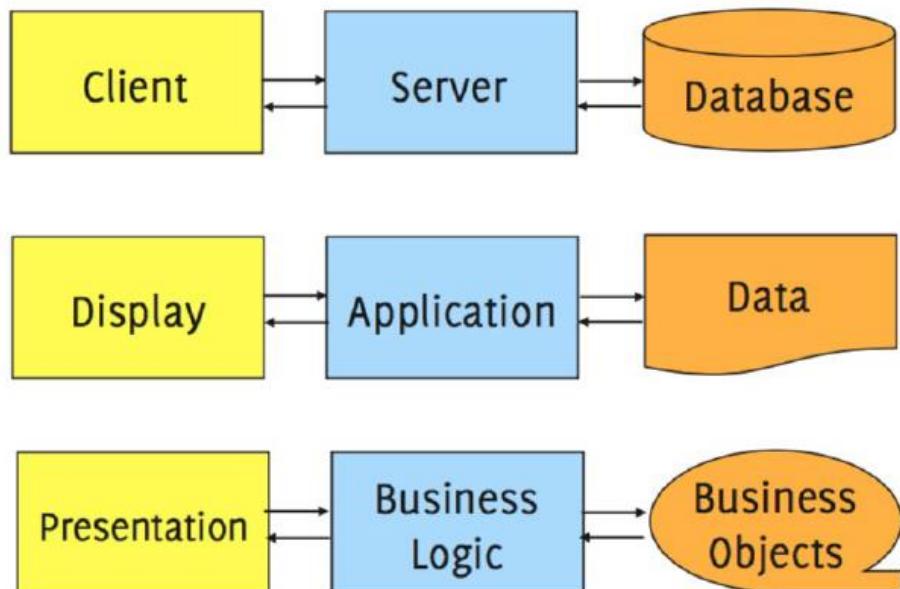
Figure 17-1. The topology of the microservices architecture style



# Architectural Styles

Separate Decision:

- the user interface (display/client)
- the business logic (server)
- the persistent state (database) of the system





# Architectural Styles Robotics Domain

## Robot as an autonomous agent

An autonomous agent is a system situated within an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future.

### Applications:

Agriculture  
Logistics  
Search & rescue

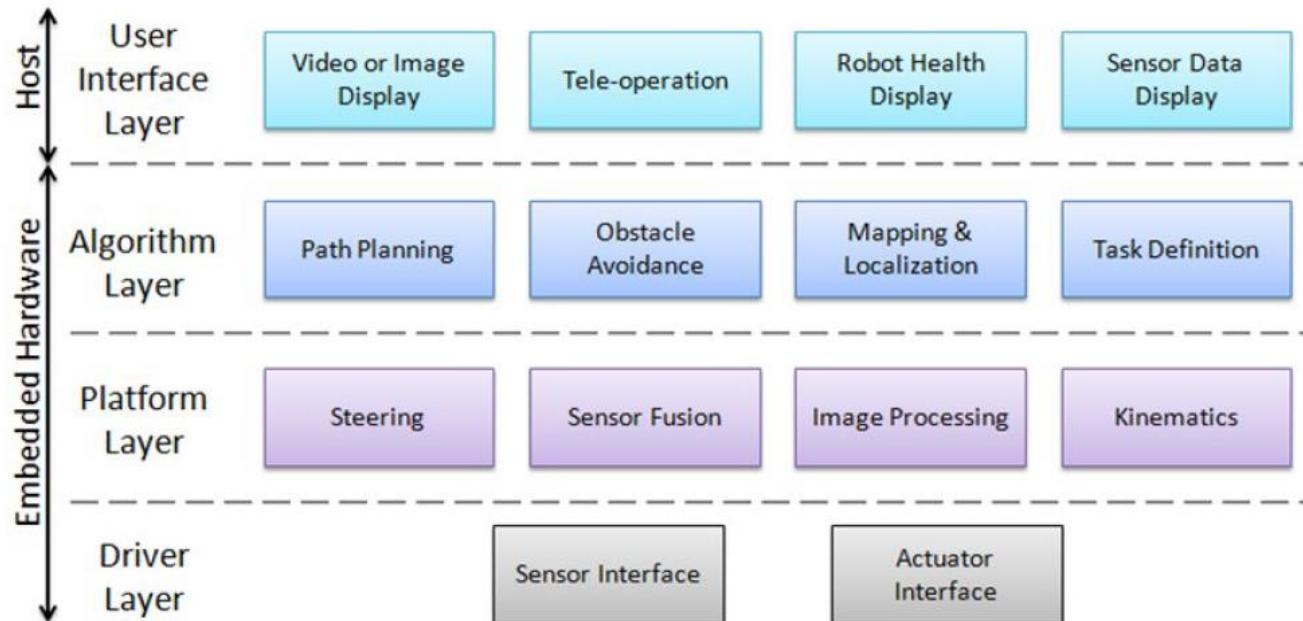
### Onboard sensors:

Camera  
Sensors (location)  
Infrared





## A Layered Approach to Designing Robot Software



Kinematics describes the motion of the bodies and deals with finding out velocities or accelerations for various objects.

An actuator is a component of a machine that is responsible for moving and controlling a mechanism or system, like motor.

# Robotics Modular Architecture

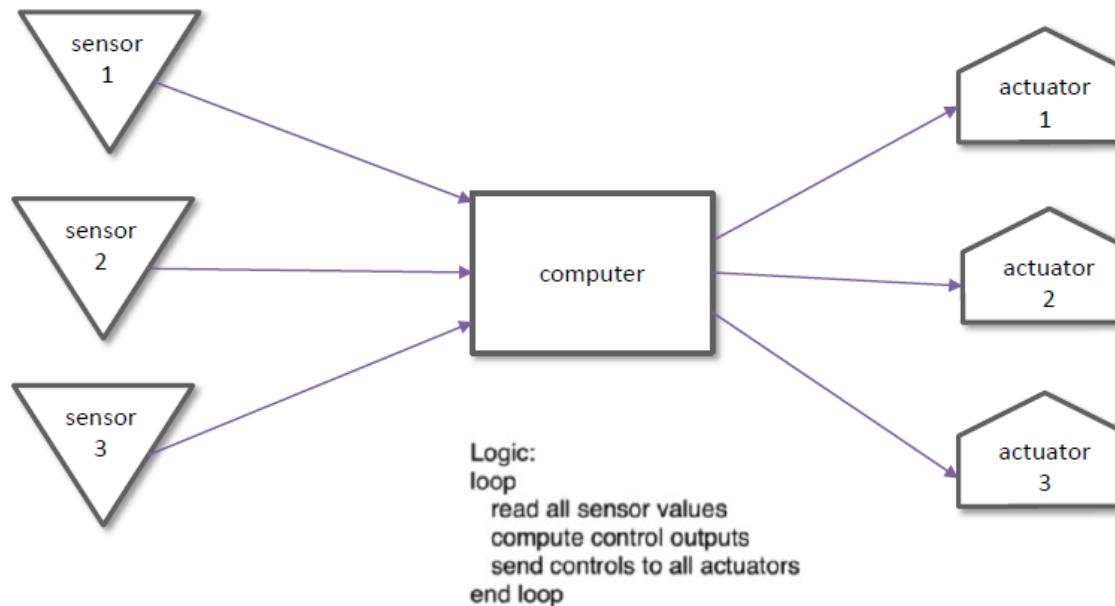
Let's Decompose the Problem:  
An autonomous robot needs to  
navigate in a complex environment

Need proper modular architecture



# Robotics Modular Architecture

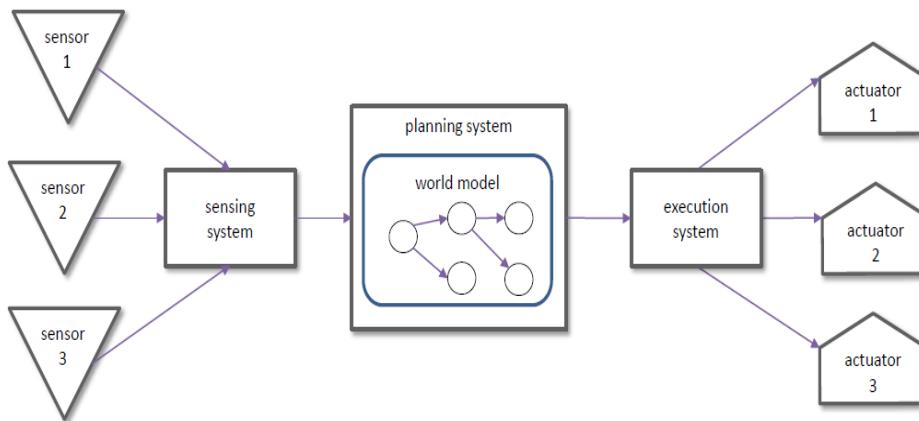
Most Simple Form: Sense, Compute, Control





# Robotics Modular Architecture

A bit more Comprehensive: Sense, Plan, Act



The job of the **sensing system** is to translate raw sensor input (usually sonar or vision data) into a world model.

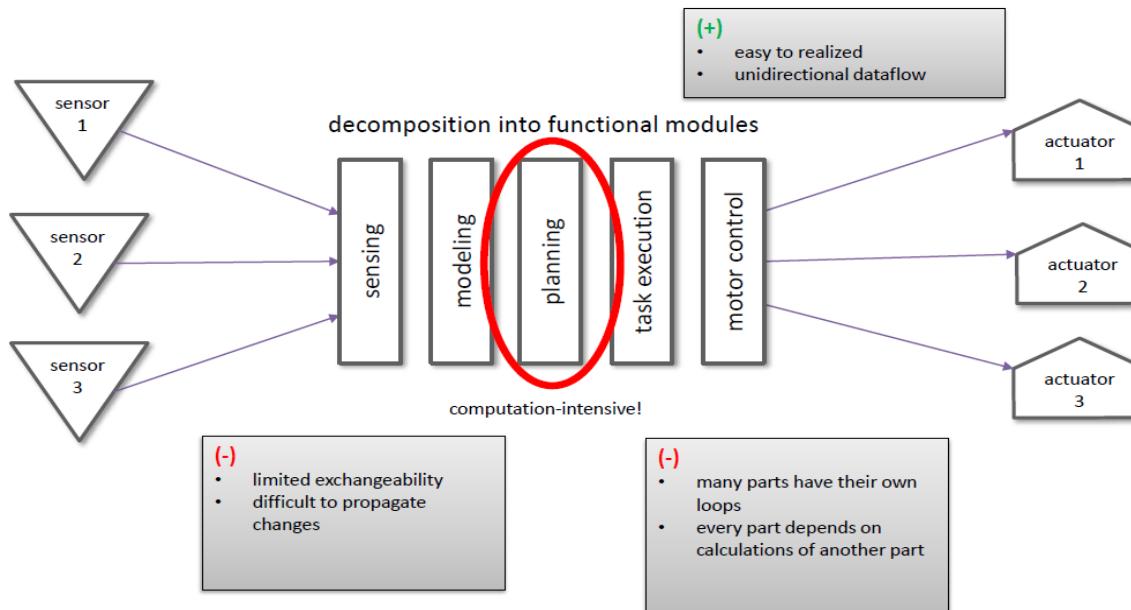
The job of the **planner** is to take the world model and a goal and generate a plan to achieve the goal.

The job of the **execution system** is to take the plan and generate the actions it prescribes.



# Robotics Modular Architecture

Primitives of Robotics: Sense, Plan, Act



The sense-plan-act (SPA) approach keeps intelligence of the system living in the planning or the programmer, not the execution mechanism.

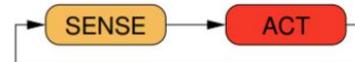


# Robotics Modular Architecture

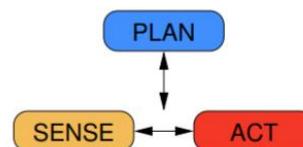
- Primitives of robotics are: **Sense**, **Plan**, and **Act**
- **Robotic paradigms** – define relationship between the primitives
- Three fundamental paradigms have proposed
  - **Hierarchical paradigm** – purely deliberative system



- **Reactive paradigm** – reactive control



- **Hybrid paradigm** – reactive and deliberative



## Disadvantages of Hierarchical Model

- Planning–Computation requirements is very slow
- The “global world” representation has to contain all information needed for planning
- Sensing and acting are always disconnected
- The “global world” representation has to be updated
- The world model used by the planner has to be frequently updated to achieve a sufficient accuracy for the particular task



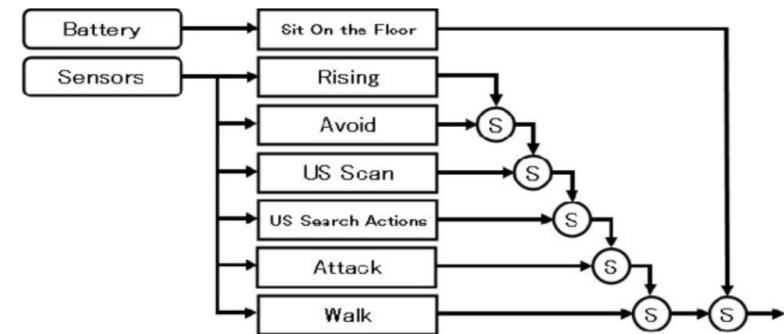
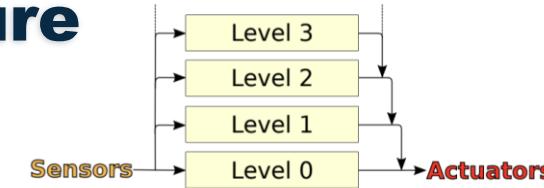
# Robotics Modular Architecture

## Subsumption architecture

Upper layers have precedence over lower layers and subsumes (absorb) the output of lower layers.

Mobile Robot can use layers of a control system for each level of competence and simply add a new layer to an existing set to move to the next higher level of overall competence.

Additional layers can be added later, to add complexity (more features)

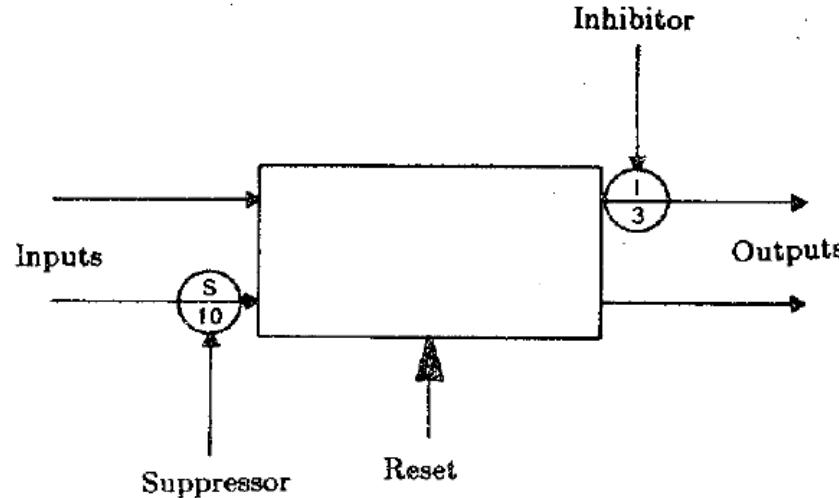


The subsumption architecture does not allow for any shared memory or other communication between the layers, making it impossible for the layers to cooperate in order to achieve a common goal



# Robotics Modular Architecture

## Subsumption architecture



A module has input and output lines. Input signals can be suppressed and replaced with the suppressing signal. Output signals can be inhibited. A module can also be reset to state NIL.

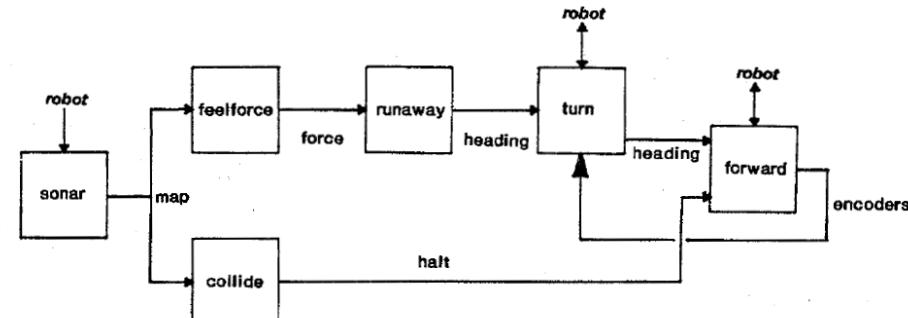


# Robotics Modular Architecture

## Subsumption architecture

The most important problem we found with the Subsumption architecture is that is it not **sufficiently modular**.

Because **upper layers interfere with the internal functions of lower-level behaviors** they cannot be designed independently and become increasingly complex.



Level 0 control system.

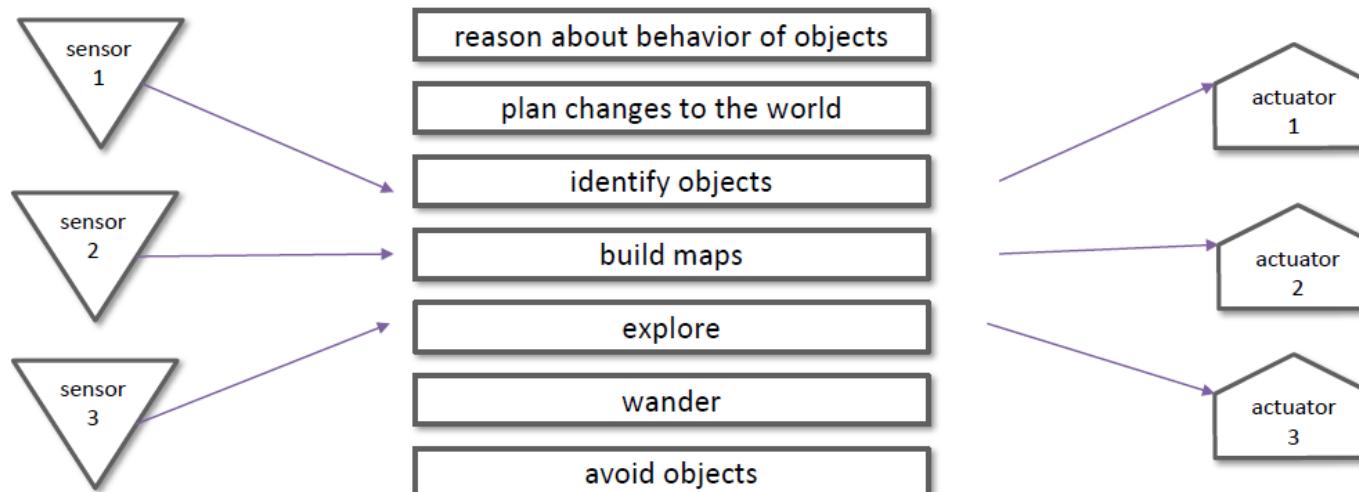
The lowest level layer of control makes sure that the robot does not come into contact with other objects. It thus achieves level 0 competence. If something approaches the robot it will move away



# Robotics Modular Architecture

Deliberative planner decomposed into features

Subsumption layers decomposed into features





# Robotics Modular Architecture

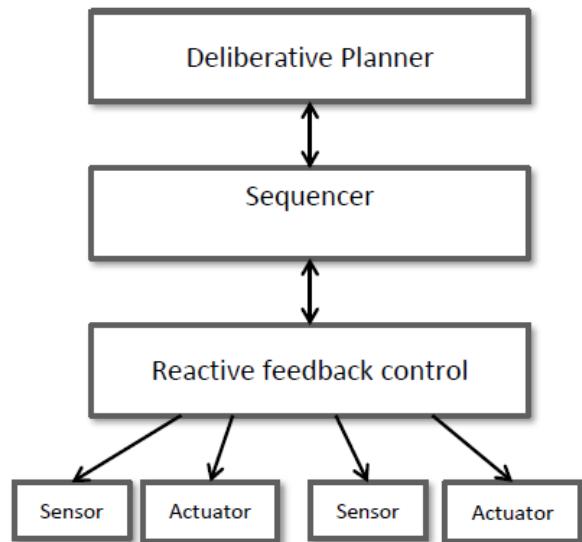
Decompose system into three layers\*

Why three layers? Different kinds of decisions

- need to plan future
- need to remember past
- need sensors input

so, three layers:

- Deliberative Planner: plans
- Sequencer: saves past
- Reactive control: stateless sensor/actuators



\* <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.43.9376&rep=rep1&type=pdf>



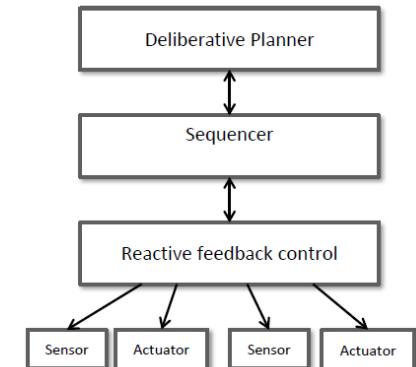
# Robotics Modular Architecture

## The Deliberator

The key architectural feature of the deliberator is that several **behavior transitions** can occur between the time a deliberative algorithm is invoked and the time it produces a result.

The pure deliberative architectures have some unresolved issues regarding symbolic representation of the agent's environment.

The deliberator can produce plans for the sequencer to execute, or it can respond to specific queries from the sequencer.

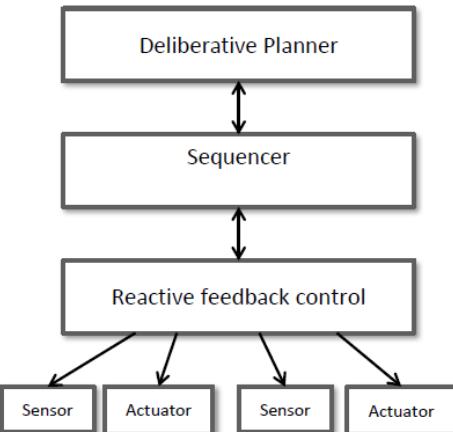




# Robotics Modular Architecture

## The Sequencer

The sequencer's job is to **select which primitive Behavior** (i.e. which transfer function) the controller should use at a given time, and to supply parameters for the Behavior.



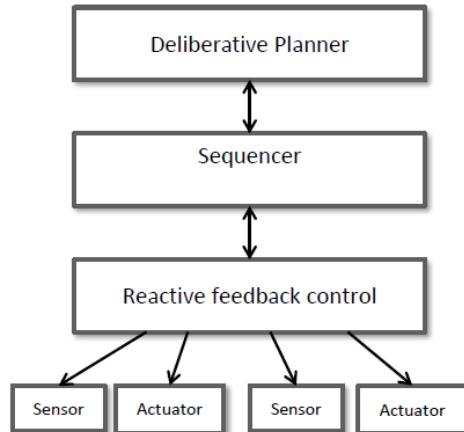
The sequencer must be able to **respond conditionally** to the current situation. One approach to the problem is to **enumerate all the possible states** the robot can be in, to use in each state.



# Robotics Modular Architecture

## The Reactive Controller Layer

The controller consists of one or more threads of computation that implement one or more feedback control loops, **tightly coupling sensors to actuators**.



Usually the controller contains a library of hand-crafted transfer functions (**called primitive behaviors or skills**).

There are several important architectural **constraints** on the algorithms that go into the controller to **provide a desired behavior**.

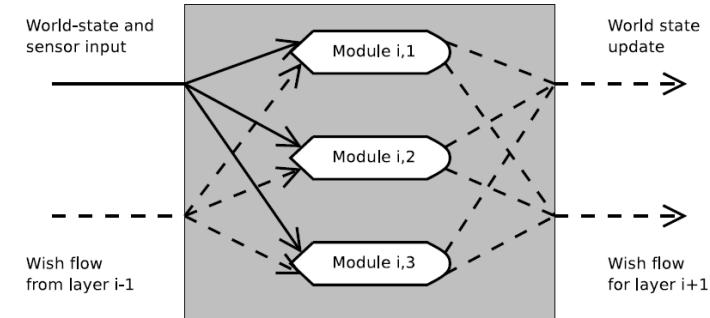


# Robotics Modular Architecture

## World state component

The world state component is a **shared memory** for the layers.

At the beginning of each turn the world state data are sent to Layer 1. The layer is able to **alter the world state** by sending an abstract wish with the desired change to the world state component. The world state component then fulfills the abstract wish by updating the world state according to the wish. The world state data are then sent to Layer 2, and so on.



A closer look at a subsumption layer showing the flow between modules inside layer-i.

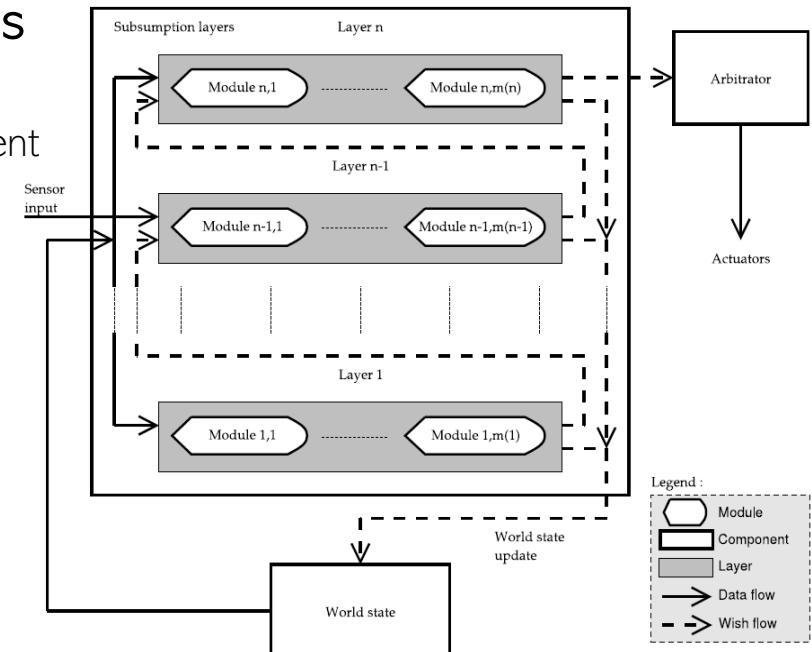
Layer-1 does not receive any wish flow.



# Robotics Modular Architecture

The hybrid architecture, shown in the Figure\* is based on three components:

- world state component, subsumption layers component and an arbitrator component.
- The architecture has two types of flow: data flow and wish flow.
- The data flow carries sensor input to the layers, data from the world state component to the layers and actions from the arbitrator to the actuators.
- The wish-flow is a special kind of flow between each layer in the architecture, from the top layer to the arbitrator and from the layers to the world state component.
- The layers create a *wish list* for the arbitrator to convert into actions. This wish list is carried by the wish flow.





# Robotics Modular Architecture

The hybrid architecture, shown in the Figure

The previous hybrid architecture modules maintain a model of Aalbot's environment in the world state component.

## Table of Enemy Information

Aalbot needs to keep information of the enemy robots in the arena. These data are kept in a table of enemy information. The table is indexed by a unique robot identifier.

For each robot, e, on the battlefield the following fields exist in the table:

- Position. The (x, y) coordinates of e at the time e was scanned.
- Energy. The energy of e at time of scan.
- Heading. The direction that e was facing at time of scan.
- Velocity. The velocity of e at time of scan. 39 40 World State Component
- Time stamp. The time at which the information was last updated, i.e. time of last radar scan of e

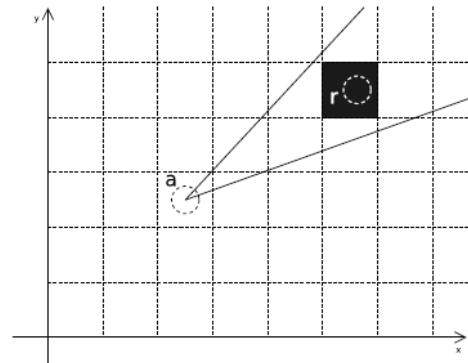


# Robotics Modular Architecture

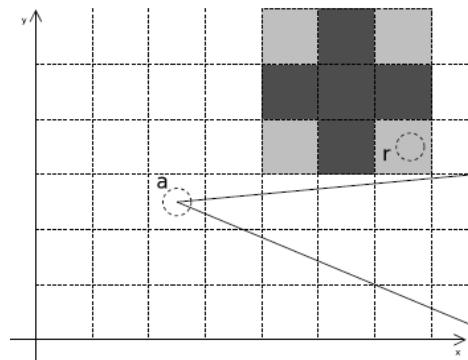
In a), Aalbot, identified by a scans a robot *r* and updates its fading memory target map.

Each square between dotted lines represent a slot, **white** squares indicate zero probability of a robot in this area, **black** squares indicate a probability of 1 that this slot holds a robot. **Grey** squares represent values between 0 and 1.

The radar cone of Aalbot is shown as the lines originating from *a*. In b), the Aalbot and robot *r* are depicted in another situation. Aalbot has moved its radar so it does not scan *r* in this turn. However Aalbot updates its target map to reflect the new possible positions of *r*.



a)



b)



# Robotics Modular Architecture

IBM's Autonomic Architecture IBM MAPE-K\* (Monitor, Analyze, Plan, Execute, Knowledge)

MARE-K consists of five main components which, form a loop, as shown in Figure:

Monitor.

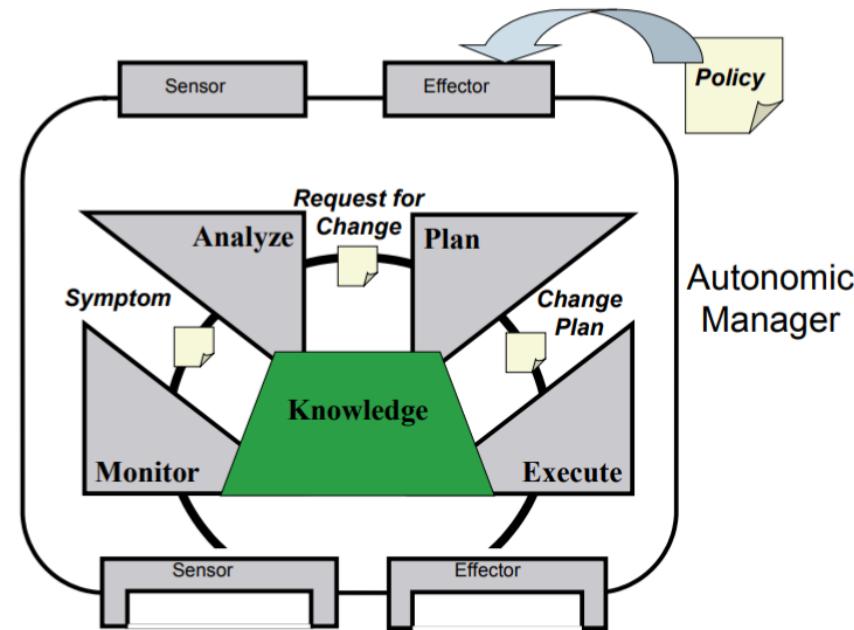
Analyze

Planning component

Execution component

Knowledge (adaptation logic)

The input to the MAPE-K architecture comes from the sensory mechanism, while the effector mechanisms carry out the action.





# Robotics Modular Architecture

IBM's Autonomic Architecture IBM MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge)

**The Monitor.** monitor the surrounding environment, including system resources.

**The Analyze component,** uses a number of algorithms to anticipate problems and possibly proffer solutions to these problems.

**The Planning component** uses the information available to the autonomic system to choose which, policies to execute.

**The Execution component,** effects the most appropriate policy/policies chosen by the system to cause a change in the physical environment



# Robotics Modular Architecture

Decentralized, Collaborative, and Autonomous Robots\*

This research developed an architecture for robot applications “Self adaptive dEcentralized Robotic Architecture” SERA\*, that **supports human-robot collaboration, as well as adaptation and coordination of single- and multi-robot systems** in a decentralized fashion.

SERA is based on layers that contain components that manage the adaptation at different levels of abstraction and communicate through well-defined interfaces.

It uses similar architecture to MAPE-K



\*"An Architecture for Decentralized, Collaborative, and Autonomous Robots" Sergio García\*, Claudio Menghi\*, Patrizio Pelliccione\*, Thorsten Berger\*, Rebekka Wohlrb\*† \*Chalmers | University of Gothenburg, Gothenburg (Sweden) † Systemite AB, Gothenburg (Sweden)



# Robotics Modular Architecture

Decentralized, Collaborative, and Autonomous Robots\*

SERA promotes engineering and integration of robotic applications developed by different vendors. It also supports Decentralization:

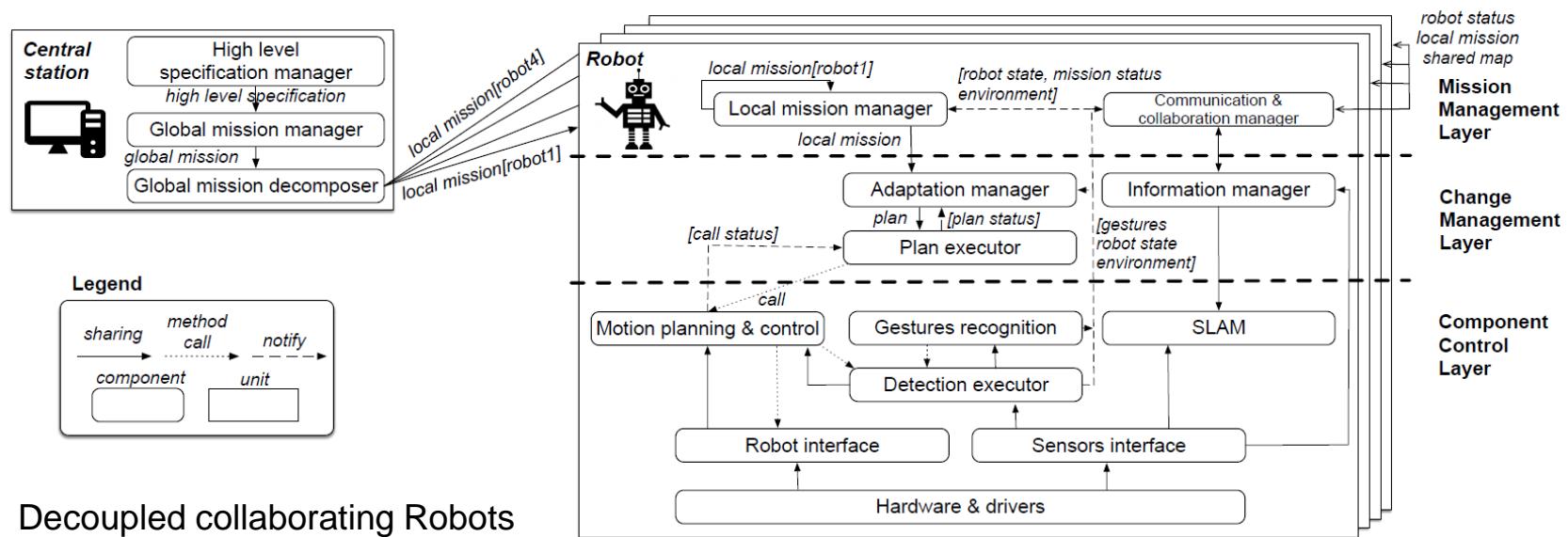
- Allows managing large teams of robots.
- Facilitates, through loose coupling, the addition, and removal of robots.
- Minimizes the cost of changes since the responsibility of **every action is within single robot that observes its environment and acts on events autonomously**.
- Facilitates data-centric workflows since data are passed directly to where they are required, at the next robot in the workflow.
- Increases robustness and decreases system vulnerability (e.g., reducing single points of failure)



# **Robotics Modular Architecture**

# Decentralized, Collaborative, and Autonomous Robots

Instance of SERA architecture of 4 collaborating Robots, Central station, and 3-Layered architectures





# Robotics Modular Architecture

Decentralized, Collaborative, and Autonomous Robots

## Central Station

Interprets high-level mission specifications, provides an interface to specify high-level missions of the application in the component. The global mission to be achieved by the whole team in a collaborative manner—checks the feasibility of the mission.

## Mission Management Layer

Local mission specifications, incorporate real-time constraints, provided to each robot by means of timed temporal logic formulae. which is compliant with their dynamic behavior.

## Component Control Layer

Provides the control actions required for the implementation of discrete paths that are generated by a high-level planner.



# Robotics Modular Architecture

This is a summary of the milestones this project has achieved and the use cases\*:

Use case 1: Achieving complex collaborative mission via decentralized control and coordination of interacting robots

Use case 2: multi-robot coordination for loading and unloading packages

Use case 3: multi-robot coordination for dynamic production assistance

Video demo can be found here: <http://www.co4robots.eu/>

\*The EU Commission's Innovation Radar highlights 9 innovations developed by Co4Robots as key for the robotics state-of-the-art! Find them by searching "Co4Robots" at: <https://www.innорадар.eu/>



# Feature-Orientation

Mapping Modules with Features



# Feature-Orientation

## Solve the problems:

- ◆ Feature Traceability
- ◆ Feature Separation
- ◆ Collaboration & Roles
- ◆ Feature Modularity

*Feature-oriented programming* is an approach for building software product lines that relies directly on the notion of features.

The idea is to decompose a system's design and code into the features it provides. This way, the structure of a system aligns with its features, ideally, one module or component per feature.

To this end, new language constructs are needed to express which parts of a program contribute to which features and to encapsulate the feature's code in composable, modular units.

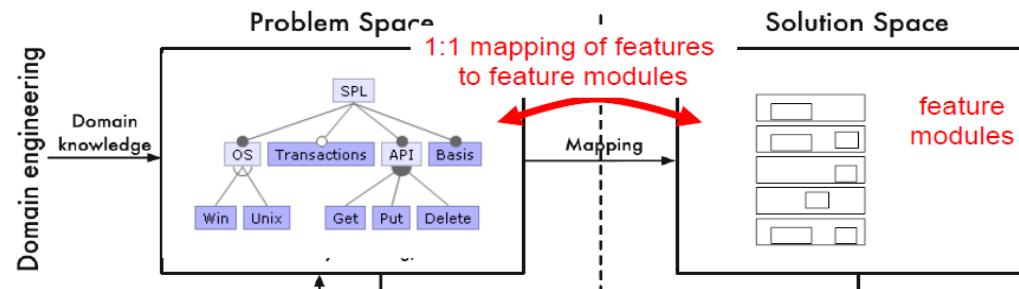


# Feature-Orientation

## Feature Traceability

*Feature traceability* is the ability to trace a feature from the problem space (feature model) to the solution space, and vice versa (its manifestation in design and code artifacts).

The whole idea of feature-orientation and feature-based product derivation depends on establishing and managing the **mapping** between the problem and the solution space, in our case, between features and their implementation artifacts.





# Feature-Orientation

## Feature Traceability

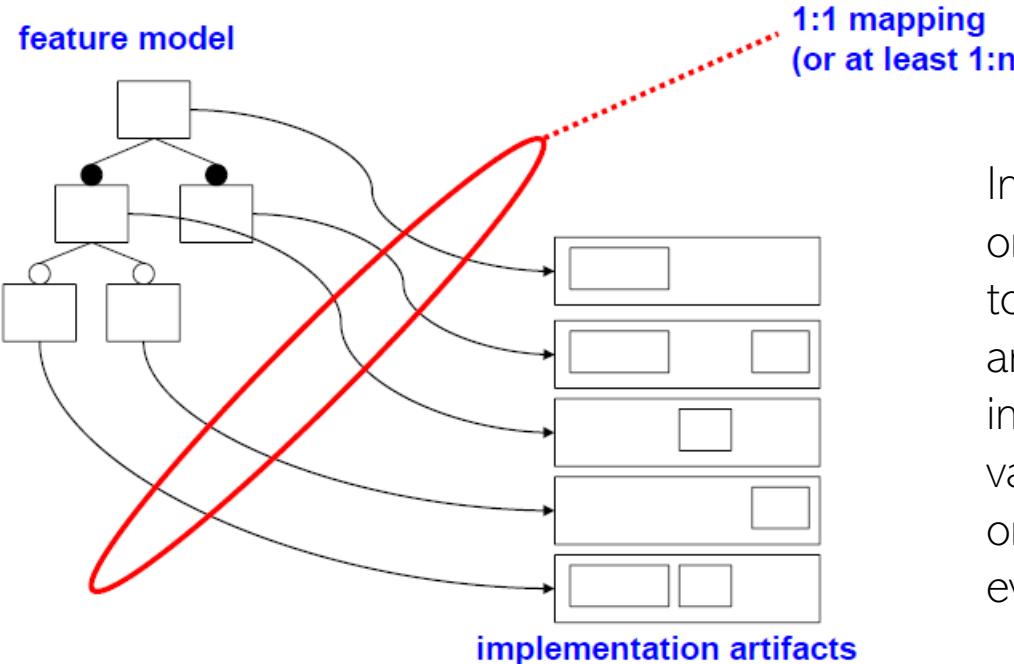
Refers to the ability to **localize all implementation artifacts** of a feature at one location in the codebase. Features are explicitly represented in the code

If feature code is not properly separated in terms of dedicated units (for example, one feature is implemented as part of many components of a system), feature traceability is impaired.



# Feature-Orientation

## Feature Traceability

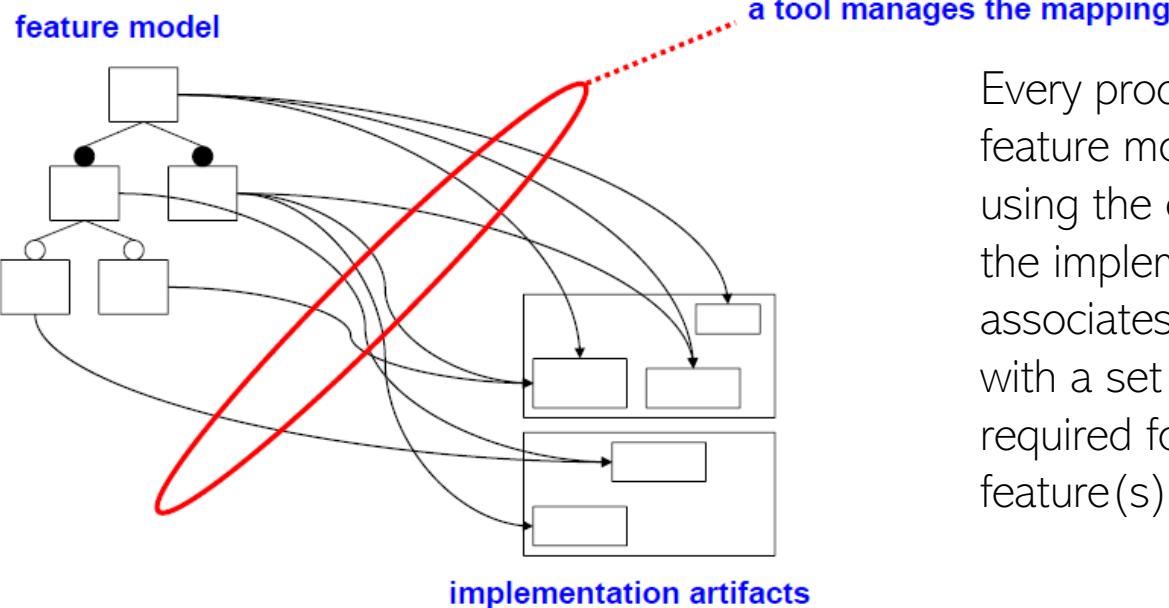


In SPL development, feature-oriented traceability is expected to map features to SPL designs and implementation elements, including commonalities and variations, to enable feature-oriented product derivation and evolution.



# Feature-Orientation

## Feature implementability



Every product represented in the feature model can be implemented using the existing assets considering the implementability relation, which associates each feature in the scope with a set of core assets that are required for implementing the feature(s).

Optimize implementability aiming for high map coverage of features over artifacts.



# Feature-Orientation

## Feature tangling

Feature tangling and scattering have **negative impacts** on the system's implementability and maintainability.

When separating features into distinct artifacts, developers can easily find all code related to that feature for maintenance or evolution tasks. **Related pieces of code are implemented together, which is known as *cohesion*.**

Due to the scattering and lack of cohesion, it can be nontrivial to trace a feature to the code fragments that are implementing it.



# Feature-Orientation

## Modularity of Feature-Oriented Programming

Language-based approach to overcome the feature traceability problem where every feature implemented through a feature module\*

- Good feature traceability
- Separation and modularization of features
- Simple feature composition

The core idea of feature-oriented programming was to use features as an additional dimension of decomposing a program.

\* Christian Prehofer. "Feature-oriented programming: A fresh look at objects." *ECOOP'97—Object-Oriented Programming*. 1997. Don Batory, Jacob Neal Sarvela, Axel Rauschmayer. "Scaling step-wise refinement." *International Conference on Software Engineering*. 2003



# Feature-Orientation

## Modularity of Feature-Oriented Programming

- When decomposing a program into features, the individual parts are typically called feature modules.
- The idea is to place everything related to a feature into a separate structure (file or folder), which is then called feature module.
- Most concepts and **tools** in feature-oriented software development follow this notion of modularity, focusing on locality and cohesion.
- In very large SPLs and features variations,  $2^f$ , then modularity of features becomes so critical, especially in agile development environment.



# Feature-Orientation

## Separation of Classes

*Features* often realized by *multiple classes*, *Classes* often realized *more than one feature*, Keep class structure, but separate classes by features.

	Classes				
	Graph	Edge	Node	Weight	Color
Search	■		■		
Directed	■	■			
Weighted	■	■		■	
Colored		■	■		■



# Feature-Orientation

## Class & Feature refinements\*

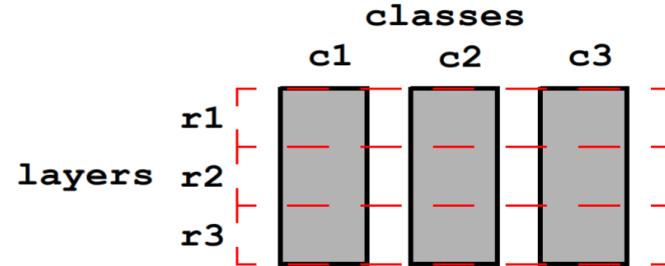
Feature refinement — module that encapsulates the implementation of a feature.

A feature refinement encapsulates not an entire method or class, but rather fragments of methods and classes.

Figure 1 depicts three classes,  $c_1—c_3$ . Refinement  $r_1$  cross-cuts these classes, i.e., it encapsulates fragments of  $c_1—c_3$ . The same holds for refinements  $r_2$  and  $r_3$ . Composing refinements  $r_1—r_3$  yields a set of fully-formed classes  $c_1—c_3$ .

Feature refinements are often called layers

In general, feature refinements are modular, though unconventional, building blocks of programs.



**Figure 1. Classes and Refinements (Layers)**



# Feature-Orientation

## Collaborations & Roles\*

*Collaboration-based design* is a fundamental technique to **decompose systems** into collaborations. A collaboration is a set of interacting classes, each class playing a distinct role, to achieve a certain function or capability.

Typically, a software system consists of multiple collaborations implementing multiple features. So, a class often participates in the implementation of multiple features.

Separating the different roles of a class as well as bundling all roles (the responsibilities a class takes in a collaboration) that belong to a collaboration are key objectives of collaboration-based design and feature-oriented programming.

\* Feature-Oriented Software Product Lines, ch.6

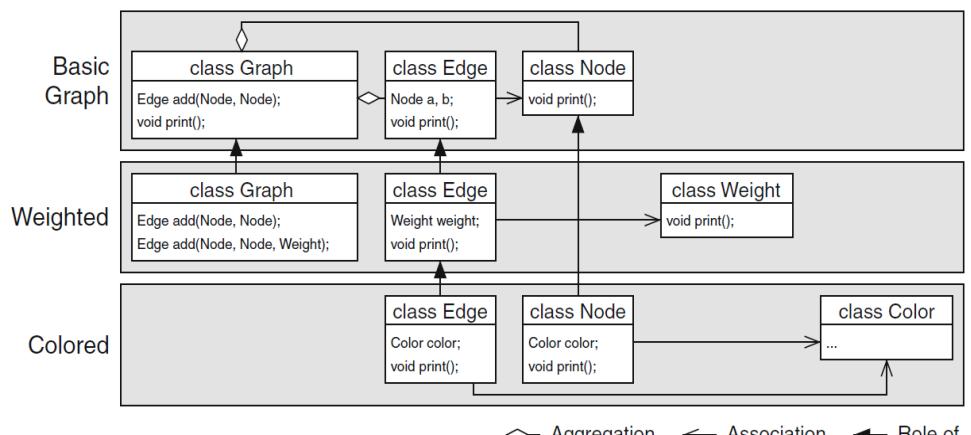


# Feature-Orientation

## Collaborations & Roles

a sample collaboration-based design is shown below, it is inspired by the graph library. The diagram uses UML-like notation with some extensions: rows (grey boxes) denote collaborations; white boxes represent classes or roles; solid arrows (that link classes column-wise) denote the application of a new role to a class.

Collaboration **BasicGraph** consists of the classes Graph, Edge, and Node, which together provide the functionality to construct and display graph structures. Collaboration **Weighted** adds roles to the classes Graph and Edge as well as a new class Weight; which extends the graph implementation to support weighted edges.





# Feature-Orientation

## Collaborations & Roles

Collaboration can be viewed as a set of classes that interact with each other to realize a feature:

- Different classes represent different *roles* within a collaboration
- A class represents different roles in different collaborations
- A role encapsulates the behavior/functionality of a class that is relevant for a collaboration

