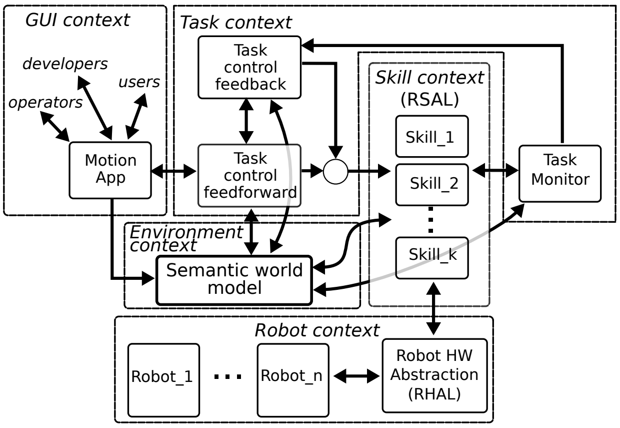
Coding with the Composition Pattern

Herman Bruyninckx
Eindhoven University of Technology / KU Leuven
http://people.mech.kuleuven.be/~bruyninc/

Embedded Motion Control
May 20, 2015

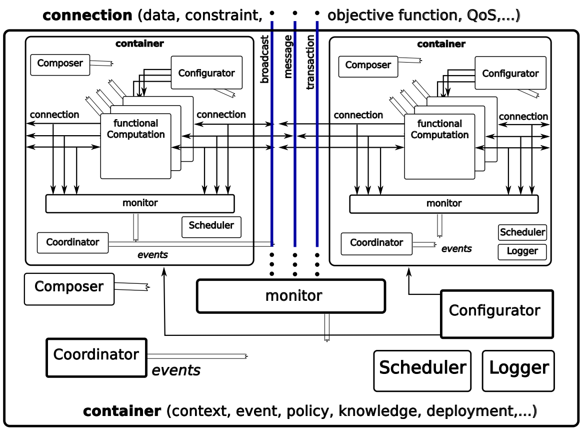
Structure for behaviour: Task-Skill-Motion



- many of these "behaviours" run *in parallel*

⇒ how do you tackle this in *your* software?

Structure for roles: Composition Pattern



- some of these "roles" are *hierarchical*

⇒ how do you tackle this in *your* software?

"Single loop" execution of roles

```
when triggered      % by OS, or other CP
do {
    communicate()   // get latest events
    coordinate()    // react to them
    configure()     // possibly requiring reconfiguration
    schedule()      // now do one's Behaviours
    coordinate()    // execution could trigger new events
    communicate()   // that others might want to know about
    log()
}
```


Example code

Online example:
http://people.mech.kuleuven.be/~lin.zhang/ecs-arduino-car

- Arduino processor

→ one single loop ("thread")

→ asynchronous IO via dedicated HW modules

This course: asynchronous IO "hidden" behind method call

- *blocking* read/write? what happens behind the screens?

→ stress test, in order to identify *platform constraints*

- from which *latency* and *jitter* does IO become *critical* disturbance for *control*?

→ `communicate()` becomes sub-system in itself:
  - (always tricky) *Inter-Process Communication*,
  - a "process-that-can-wait" architecture,
  - (de)multiplexing all IO in one "process message"

Next question to answer: one thread app?

What tasks/behaviours does your app execute:

- sensing?
- world modelling?
- planning?
- control? (discrete & continuous)

Can they all be serialized?

- can your app tolerate that `Task-A` be *delayed* by `Task-B`?
- if so, what is "right" order, *inside main "loop"*?
  what is "right order" inside `Task-A`?
- if not, how many "processes" do you need?
  what are their inter-process communication (IPC) *needs*?
  what IPC *mechanisms* do you know/need?

Task-X loop template

when `scheduled` **do** { `act(); prepare();` }

with

```
act() {
  sense();        // get sensing data out of "process message"
  control();      // get continuous part in "process message"
  communicate();  // to get control out as fast as possible
}

prepare() {
    world-model-update();
    plan(); // compute feedforward for next loop
    ...
    if monitor() then {coordinate(); configure();}
}
```


Main loop template for multiple tasks in one single thread

```
when triggered do {
  communicate()        // get "process message" and
                       // deserialize for each Task
  coordinate()         // react to app-level events
  configure()          // possibly requiring reconfiguration
  schedule-acts()      // now do all Tasks' act()
  communicate()        // serialize all Tasks' control
                       // and get "process message" out!
  schedule-prepares()  // now do all Tasks' prepare()
  coordinate()         // execution could trigger new events
  communicate()        // "process message" with app events
  log()
}
```


Summary

- control applications have **a lot of structure**

⇒ exploit it, for *efficiency*, *readability* and *composability*

- **priorities** between tasks is often needed

⇒ do it by *your own* scheduling, not the OS's!
(because the priorities are often time and context dependent. . .)

- main gain in *control performance* comes from **separate scheduling** of **act()** and **prepare()** of "parallel" behaviours

⇒ impossible with "one-behaviour-in-one-process" design!

- real multi-threaded/multi-processing/multi-node control often becomes *a lot* more complex, due to **overzealous** drive *to keep data* **consistent** over all threads, processes, nodes. . .