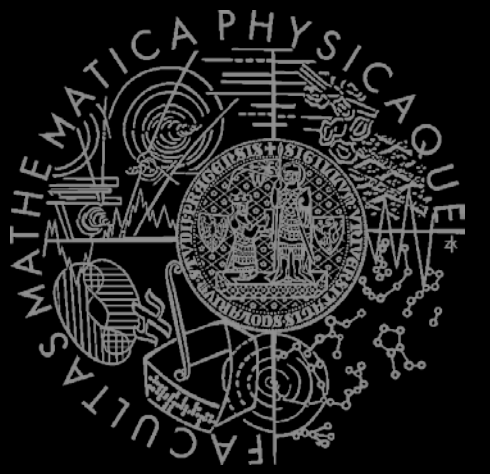
Faculty of Mathematics and Physics
Charles University in Prague
28th April 2015



UT2004 bots made easy!

Pogamut 3

Lecture 9 –yaPOSH

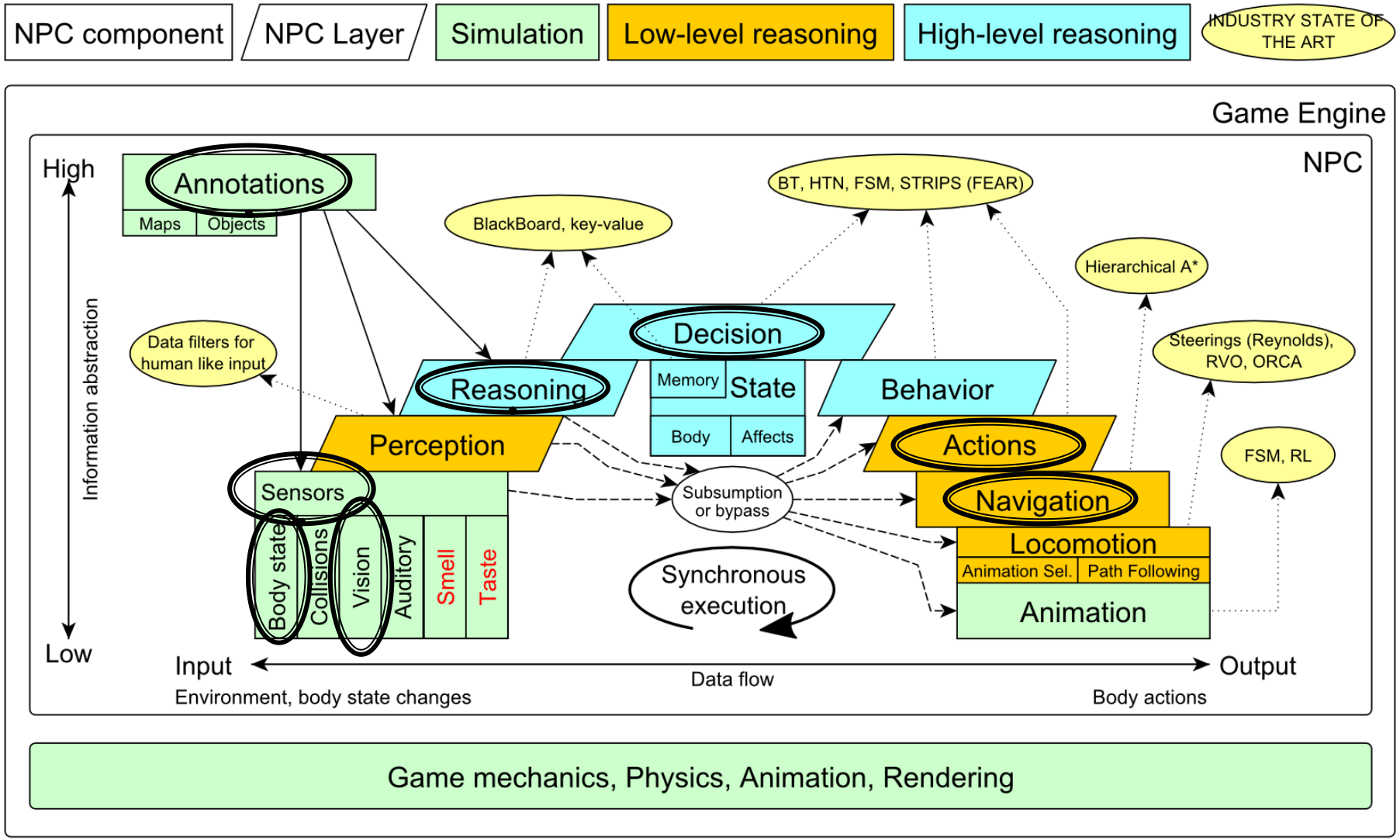
Warm Up!

- Fill the short test for this lessons
  - 8 minutes limit
  - **http://alturl.com/zgu9b**
  - https://docs.google.com/forms/d/1MT1-Xi6YBWtFSrtKQJE-7su2O_-h6BoUhuLnGwiO-Hk/viewform

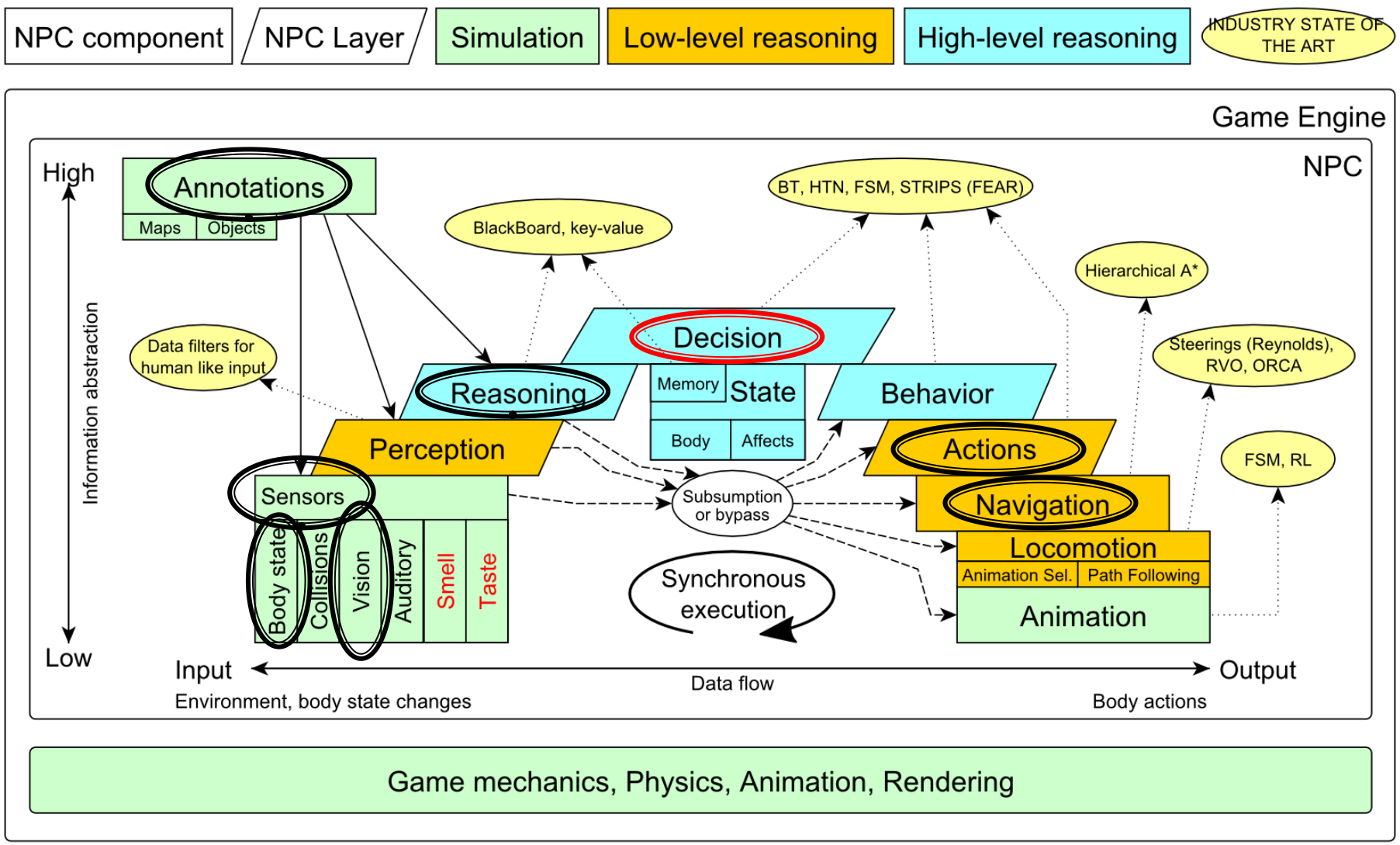
Today's menu

1. **Big Picture**
2. BOD & POSH
3. yaPOSH
4. Simple DMBot in yaPOSH

Big Picture
Already covered



Big Picture
Today



Today's menu

1. Big Picture
2. **BOD & POSH**
3. yaPOSH
4. Simple DMBot in yaPOSH

Behavior Oriented Design
Methodology

- **BOD** (Behavior Oriented Design)
  - A methodology for developing control of complex intelligent agents
    - virtual reality characters, humanoid robots or intelligent environments…
- Combines the advantages of Behavior-Based AI and Object Oriented Design.

- Authored by Joanna J. Bryson
  - http://www.cs.bath.ac.uk/~jjb/web/bod.html

Behaviot Oriented Design

Intelligence by design

Behavior Oriented Design

by Joanna J. Bryson (UK)
http://www.cs.bath.ac.uk/~jjb/web/bod.html

1. Specify top-level decision
   a) Name the behaviors that the bot should do
   b) Identify the list of sensors that is required to perform the behavior
   c) Identify the priorities of behaviors
   d) Identify behavior switching conditions
2. Recursion on respective behaviors until primitive actions reached

Behavior Oriented Design

BOD in human language

1. State the goal of you agent behavior
   - E.g. It will be a Deathmatch bot
2. Brainstorm what it will mean to fulfill the behavior goal
   - E.g. fight players, gather items
3. Think about conditions that should be fulfilled for the respective behaviors
   - E.g. I'll fight only when I see enemy and have proper weapon
4. Revise, revise, revise
   - Oh wait, what if I don't have the proper weapon, I should add a behavior to flee from fight and gather some weapon.
5. Pick one of the specified top level behaviors and apply recursion from point 1!
6. When you end up with sufficiently simple and clear defined sense/action – **NAME IT WELL**, implement it and test it!

Behavior Oriented Design
Iterative Development

Recursion == Iterative development

1. Select a part of the plan to extend next.
2. Extend the agent with that implementation
   - Extend the plan, code actions and senses
   - Test and debug that code (!!!)
3. Revise the current specification.

Behavior Oriented Design
Revising BOD Specifications

- Name the behaviors (functions) logically!
  - Good method name is better than documentation!
- Reduce code redundancy
  - Use copy paste with caution or not at all!
- *Avoid Complex Conditions*
  - The shorter condition, the better the understanding
- *Avoid Too Many If-then rules at one level*
  - One level of decision making usually needs no more than 5 to 7 if-then rules, they may contain fewer..
- *When in doubt, favor simplicity.*

POSH?

- **POSH**
  - Parallel-rooted, Ordered Slip-stack Hierarchical planner
- To put it simply:
  - a reactive planner working with **FIXED**, **PRE-SET** plans
- To put it simpler:
  - a tool enabling to specify **if – then** rules with **priority** in a **tree like structure**
- Advantage:
  - Makes you think about the behavior in human terms more than the code
- There are multiple POSH implementations
  - POSH, pyPOSH, JavaPOSH, yaPOSH, …
  - Their language varies a lot

POSH
Control Structures I

- General structure of the POSH "tree"
  - Root is a Drive Collection
  - Root's children are Drives
  - Drive child is either Competences, or Action Pattern or Action
  - Competence children are again either Competences, or Action Pattern or Action
  - Almost every node has associated a "triggering/goal" condition

[Bryson, 2001]

POSH
Control Structures II

- **Action Pattern**
  - $(a_1, a_2, \ldots, a_n)$ a sequence of actions
  - e.g., "baa" and look at it (sheep)
- **Competence**:
  - $\{s_1, \ldots, s_n\}$ a set of competence steps
  - steps that can be performed in different orders (i.e., a set of sequences)
  - one of the steps can be a goal step
  - the competence returns a value: **DONE** if the goal is accomplished, **RUNNING** if none of its steps fire
- **Competence step**
  - **<p, r, a, [n]>**
  - a priority, a releaser, an action, a number of retries
  - the action can also be a competence / action pattern

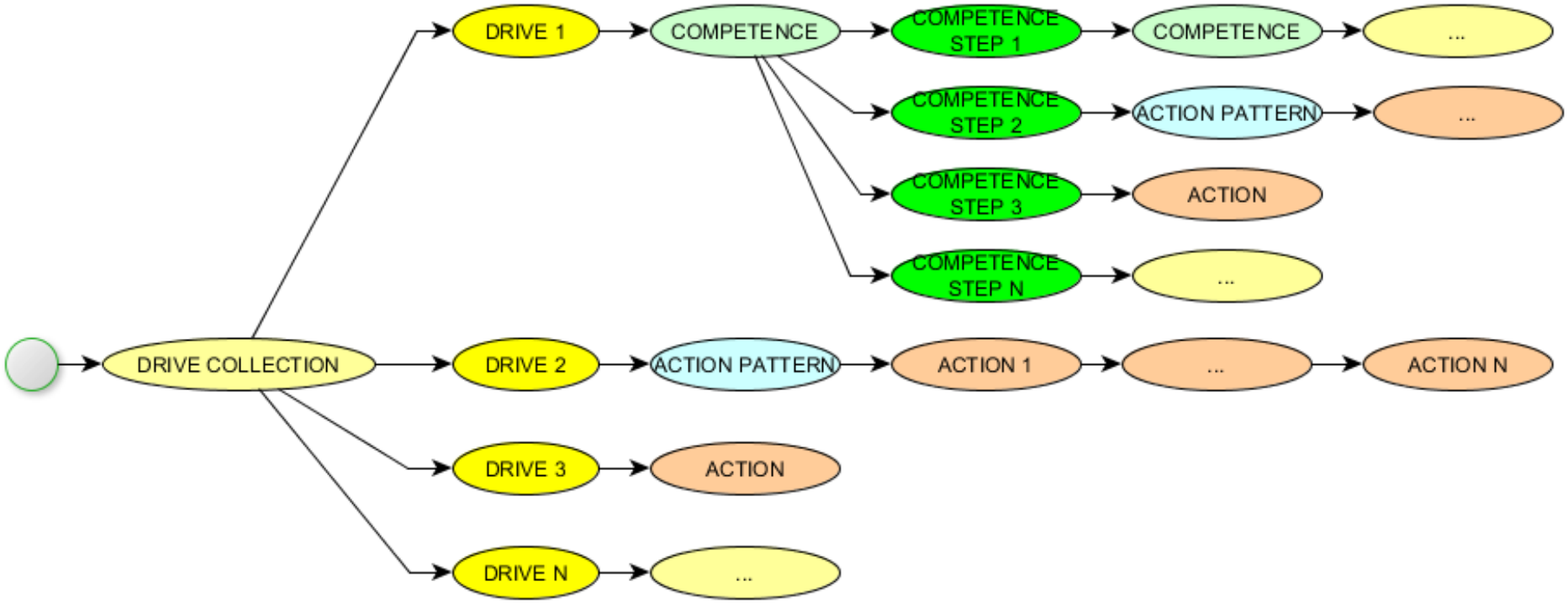
[Bryson, 2001]

POSH
Control Structures III

- **Drive Collection**
  - $\{ d_1, \ldots, d_n \}$ d is a drive element
  - the root of the hierarchy
  - a drive element: <p, r, a, A, [f]>
    - *p* – a priority
    - *r* – a releaser
    - *a* – a currently active element of the drive element (a sub-element)
    - *A* – the top element (i.e., a collection, action pattern, or an action) of the drive element → slip-stack
    - *f* – a maximum frequency at which this drive element is visited
      - e.g., jump every five seconds
  - for any cycle of the action selection, only the drive collection itself and at most one other POSH element will have their releasers examined
- One drive element can suspend temporarily another drive element
  - a competence step cannot interrupt another competence step
- When the suspending drive element terminates, the suspended drive element continues

POSH

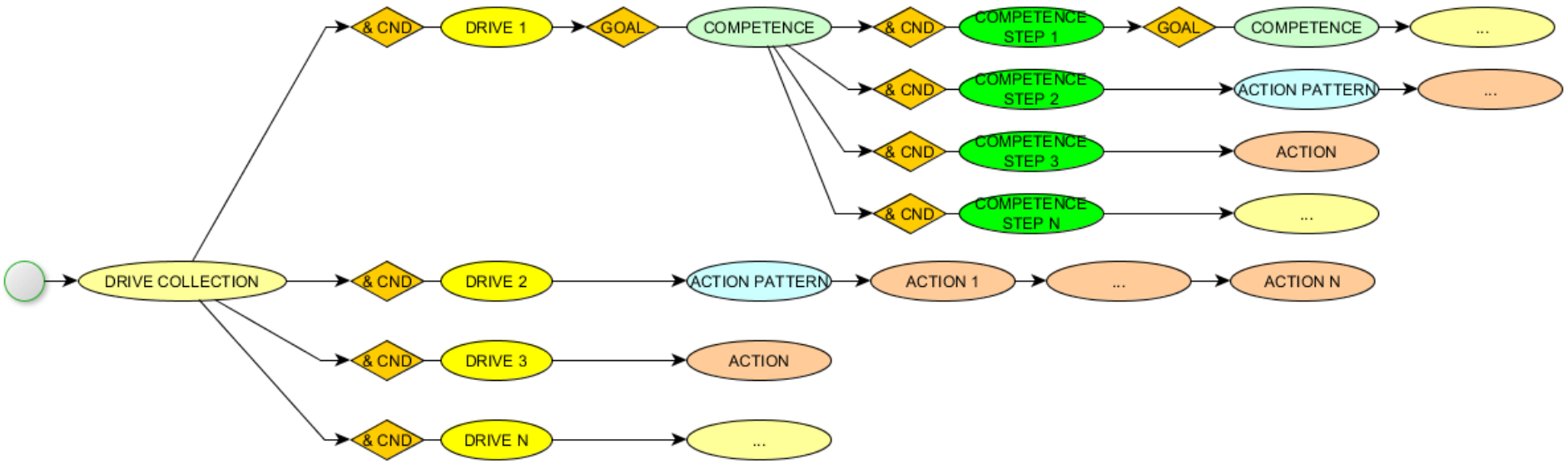
Control Structures I - Visualization



- POSH defines Tree-like structure

POSH

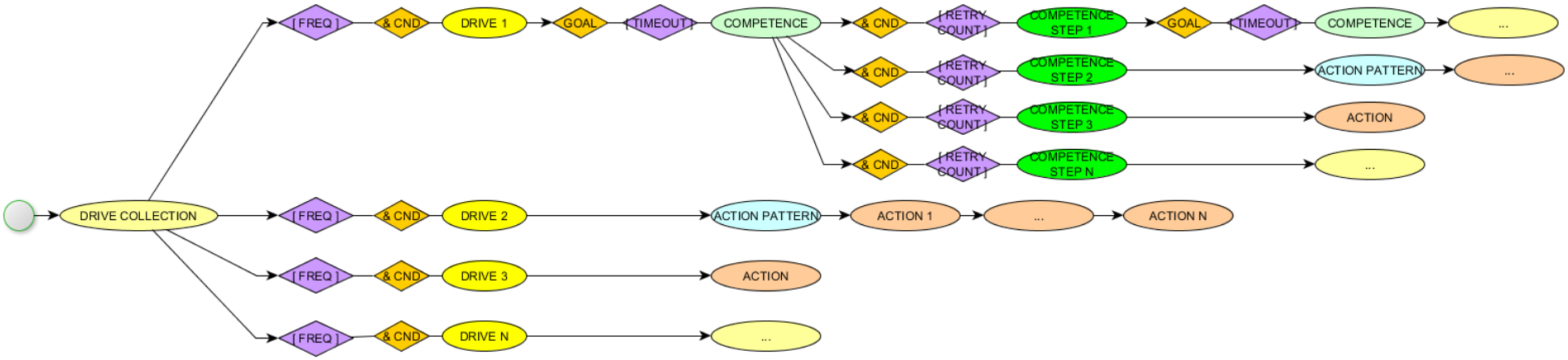
Control Structures II - Visualization



- Respective DRIVEs and COMPETENCE STEPs has TRIGGERINIG conditions
- COMPETENCEs has GOAL condition

POSH

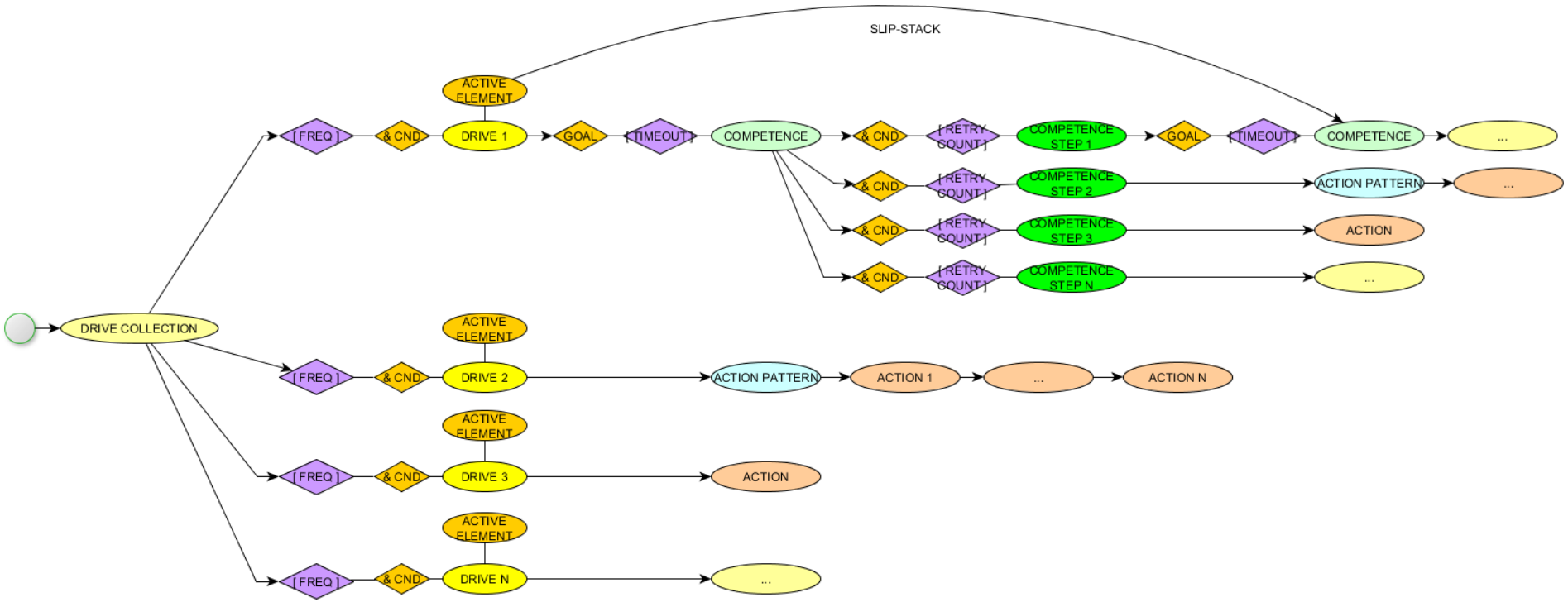
Control Structures III - Visualization



- DRIVEs may have option FREQUENCY decorator
- COMPETENCEs may have TIMEOUT decorator
- COMPETENCE STEPs may have RETRY-COUNT decorator

POSH

Control Structures IV – Parallel + Slip Stack



- Each DRIVE has an ACTIVE (TOP) ELEMENT that gets executed when DRIVE triggers
- DRIVE stack (and ACTIVE ELEMENT) is RESET if SWITCHING occurs
- **Multiple DRIVEs can run in parallel (if defined as such)**

PyPOSH

```
def init_senses( self ):
    self.add_sense( "see-player", self.see_player )
    ...

def init_acts( self ):
    self.add_act( "move-player", self.move-player )
    ...

def see_player( self ):
    ...
```

Python

top-level

checking period / frequency

```
(RDC life (goal( (fail) ) )
    ( drives
  prio: 1 (( hit( trigger( * (hit-object)(is-rotating False) ) ) avoid ))
        2 (( follow( trigger( (see-player) ) ) follow-player ))
        3 (( wander( trigger( (succeed) ) ) wander-around ))
    ) )
```

timeout condition *terminate (goal) condition*

```
(C wander-around (minutes 10) (goal( (see-player) ) )
    ( elements
        (( close-enough( trigger( (close-to-player) ) ) stop-bot ))
        (( move( trigger( (see-player) ) ) move-player ))
    ) )
```

if *then*

"Lisp"

PyPOSH

```
def init_senses( self ):
    self.add_sense( "see-player", self.see_player )
    ...

def init_acts( self ):
    self.add_act( "move-player", self.move-player )
    ...

def see_player( self ):
    ...


(RDC life (goal( (fail) ) )
    ( drives
        (( hit( trigger( (hit-object)(is-rotating False) ) ) avoid ))
        (( follow( trigger( (see-player) ) ) follow-player ))
        (( wander( trigger( (succeed) ) ) wander-around ))
    ) )

(C wander-around (minutes 10) (goal( (see-player) ) )
    ( elements
        (( close-enough( trigger( (close-to-player) ) ) stop-bot ))
        (( move( trigger( (see-player) ) ) move-player ))
    ) )
```

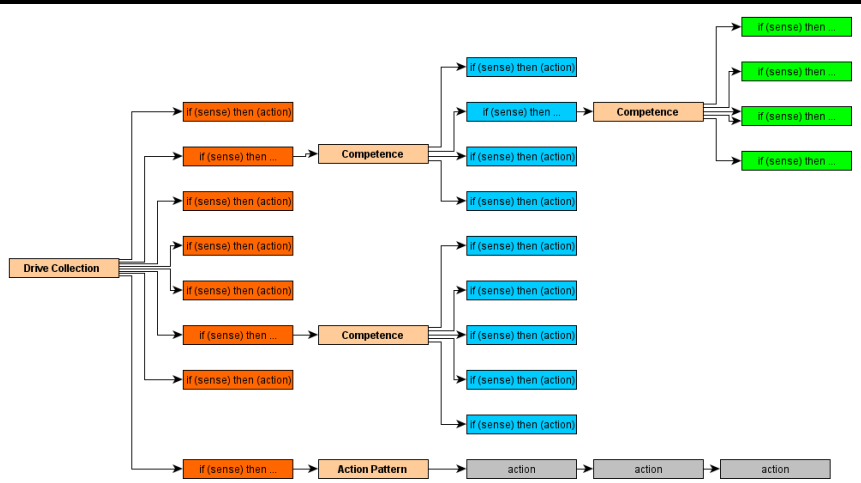
Practice Lesson
Outline

1. Big Picture
2. BOD & POSH
3. **yaPOSH**
4. Simple DeathMatch Bot in yaPOSH

yaPOSH
Introduction

- **yaPOSH**
  - **y**et-**a**nother **P**arallel-rooted, **O**rdered **S**lip-stack **H**ierarchical planner
- To put it simply:
  - a reactive planner working with **FIXED**, **PRE-SET** plans
- To put it even simpler:
  - a tool enabling to specify **if – then** rules with **priority** in a **tree like structure**
- Advantage:
  - Makes you think about the behavior in human terms more than the code

yaPOSH
Primitives



- **Actions and Senses**
  - if (sense) then (action)
- **Drive Collection (DC)**
  - First level of if-then rules
- **Competence (C)**
  - Second – Nth level of if-then rules
- **Action Patterns (AP)**
  - Specifies N actions that will be performed in a sequence

yaPOSH
Differences from POSH

- **Actions (and all nodes)**
  - Are DURATIVE
    - Returns FINISHED, RUNNING, RUNNING_ONCE, FAILED
- **Drives**
  - No timeout decorator, No slip-stack, No parallelism
- **Competence**
  - No goal node (just another inner-reusable node)
- **Competence Step**
  - No retry count decorator

yaPOSH
Plan structure (Java glasses)


```
DriveCollection / Competence {
    1. if (sense1()) then competence1(); return;
    2. if (sense2()) then competence2(); return;
    3. if (sense3()) then action-pattern1(); return;
    4. if (sense4()) then competence3(); {
            1. if (sense5()) then action1(); return;
            2. if (sense6()) then competence4(); return;
            3. if (sense7()) then action2(); return;
            4. if (sense8()) then action-pattern(); return;
               …
            N. return;
    }
    …
}
```

DriveCollection is the root of if-then tree of rules.
Competence is another level of if-then tree of rules.

yaPOSH
Plan structure (Java glasses)


```
ActionPattern.run() {
     while (!action1-finished()) { action1(); }
     while (!action2-finished()) { action2(); }
     while (!action3-finished()) { action3(); }
}
```

ActionPattern is sequence of action.

yaPOSH
Plan structure (Java glasses)


```
ActionPattern.run() {
     if (this.step == 1) {
        while (!action1-finished()) { action1(); }
        this.step = 2;
     }
     if (this.step == 2) {
        while (!action2-finished()) { action2(); }
        this.step = 3;
     }
     if (this.step == 3) {
        while (!action3-finished()) {action3();}
        this.step = 1; // reset
        return ActionResult.FINISHED;
     }
}
```

ActionPattern is tracking the step it executed last and always continues from that one.

yaPOSH
Plan structure (Java glasses)


```
ActionPattern.run() {
     if (this.step == 1) {
        while (!action1-finished()) { action1(); }
        if (action1-failed()) { this.step = 1; return ActionResult.FAILED; }
        this.step = 2;
     }
     if (this.step == 2) {
        while (!action2-finished()) { action2(); }
        if (action2-failed()) { this.step = 1; return ActionResult.FAILED; }
        this.step = 3;
     }
     if (this.step == 3) {
        while (!action3-finished()) {action3();}
        if (action3-failed()) { this.step = 1; return ActionResult.FAILED; }
        this.reset();
        return ActionResult.FINISHED;
     }
}
```

Of course, **ActionPattern** honors FAILED state, which resets it.

yaPOSH

Plan structure (Java glasses)

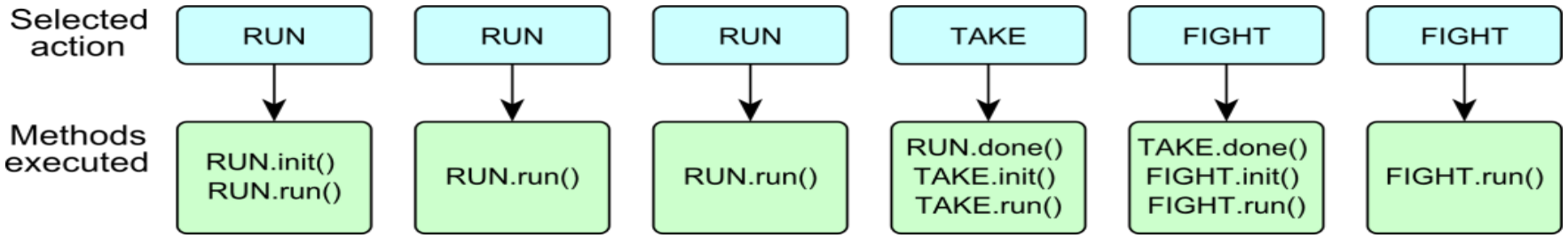

```
ActionPattern.run() {
     if (step >= children.size()) {
        this.reset();
        return FINISHED;
     }
     Node child = children[step];
     switch (child.run()) {
        case FINISHED:
                ++step;
                return this.run();
        case RUNNING:
                return RUNNING;
        case RUNNING_ONCE:
                ++step;
                return RUNNING_ONCE;
        case FAILED:
                this.reset();
                return FAILED;

     }
}

ActionPattern.reset() {
     step = 0;
     for (Node child : children) child.reset();
}
```

yaPOSH
Senses

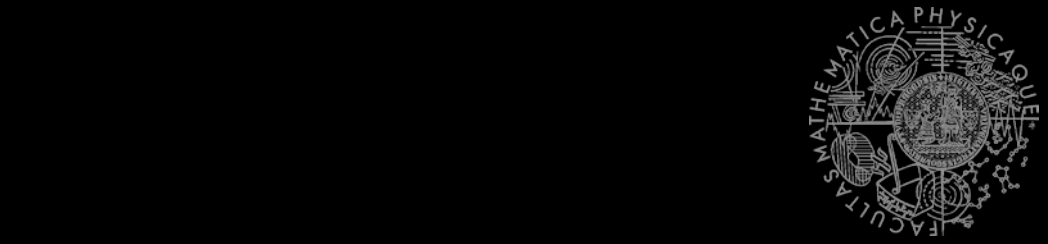
- Senses
  - Represent condition (Do I see a player?)
  - Return basic types
    - Boolean, Integer, Double, String, …
  - Can be queried either as ==, !=, >, <, <= or >=
  - E.g.:

yaPOSH
Action execution

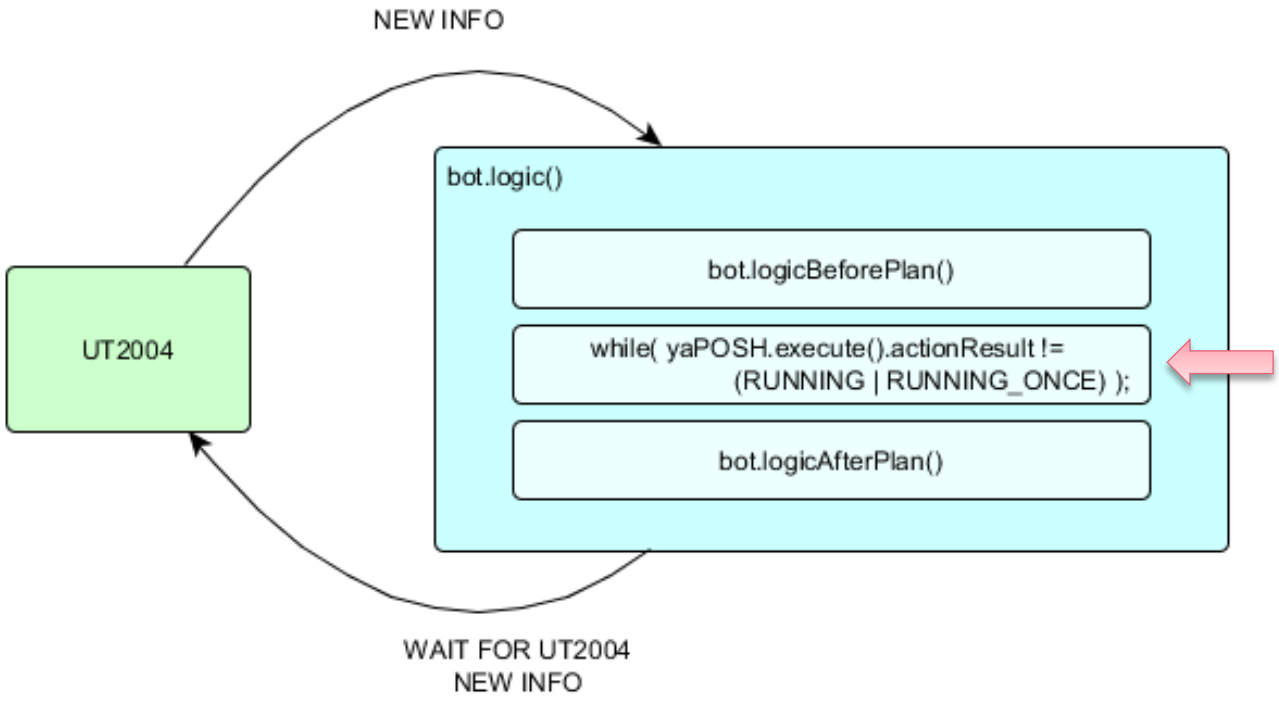
- Actions
  - Represent an action in the environment
  - Are expected to return:
    - **FINISHED** (an action has been finished successfully),
    - **RUNNING** (an IVA action is still being executed within the environment),
    - **FAILED** (an action execution has failed).
  - Have three methods – **init(), running(), done()**



yaPOSH
Execution semantics



- yaPOSH runs within “logic()” method you know from standard Java bot

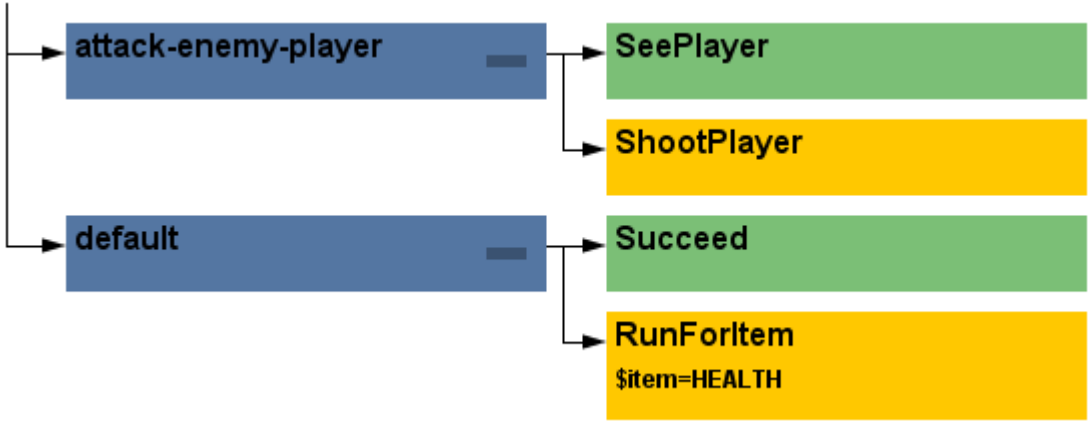


yaPOSH will execute until it finds an action that affect the environment and requires new info From the environment.

Bad plan may stall the bot!

yaPOSH
Plan structure (the real)

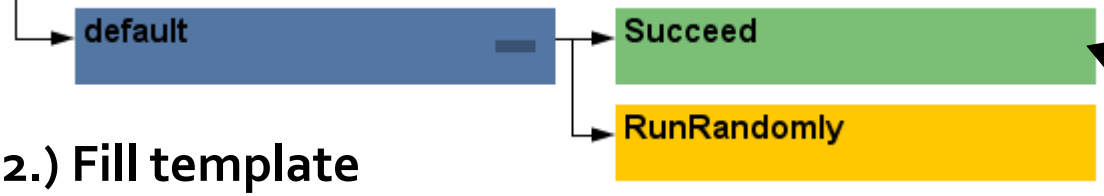

```
(
        (C heal
                (elements
                        ((stop-shoot (trigger ((duelbot.sense.IsShooting))) duelbot.action.StopShooting))
                        ((run-medkit duelbot.action.RunForItem($item='duelbot.Item.HEALTH)))
                )
        )
        (DC life
                (drives
                        ((attack-enemy-player (trigger ((duelbot.action.SeePlayer))) duelbot.action.ShootPlayer))
                        ((default (trigger ((cz.cuni.amis.pogamut.sposh.executor.Succeed))) duelbot.action.RunForItem($item='duelbot.Item.HEALTH)))
                )
        )
)
```

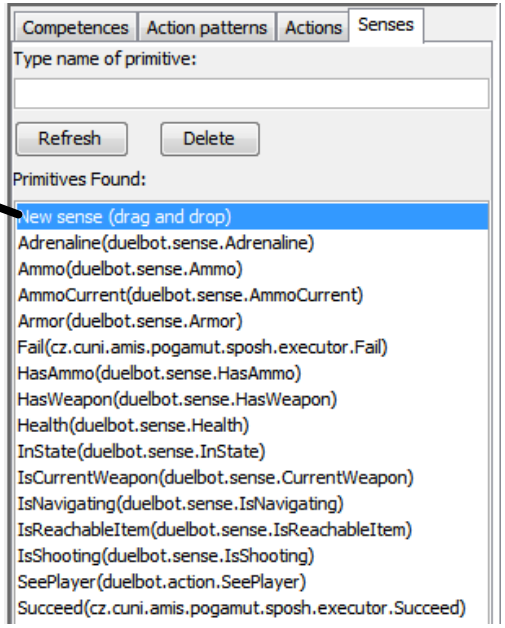


yaPOSH

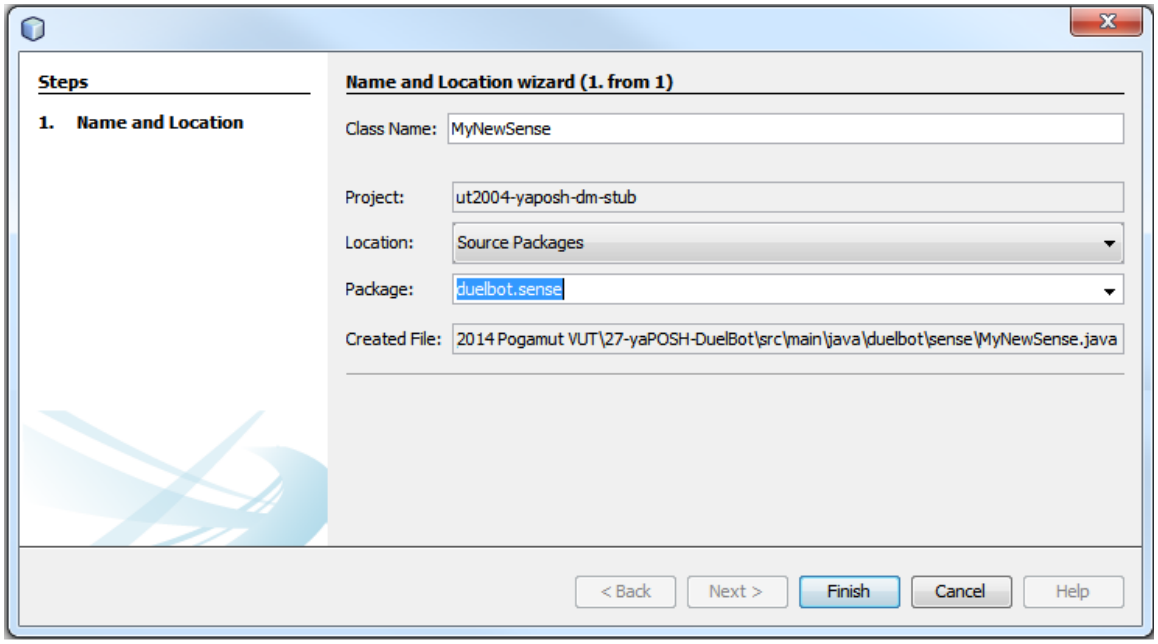
How to make new Sense?

1.) DRAG & DROP!





2.) Fill template



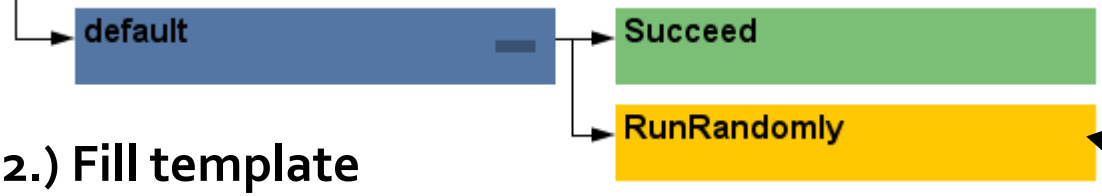
3.) Edit generated Java source file

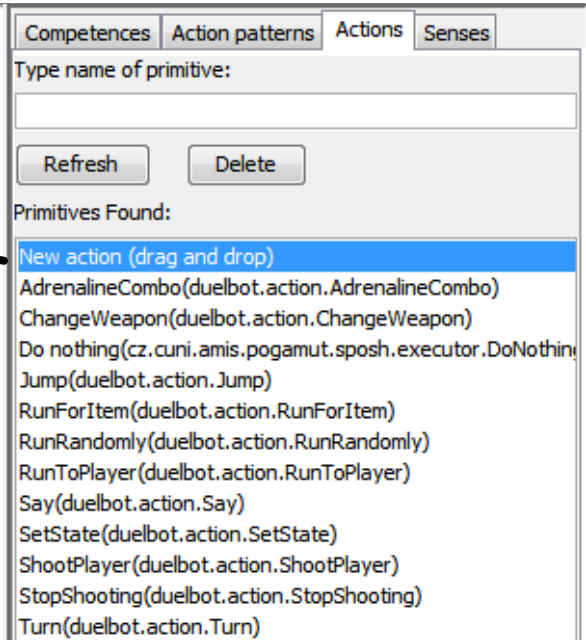
```
@PrimitiveInfo(name = "Can see a player", description = "Do I see a player?")
public class SeePlayer extends ParamsSense<AttackBotContext, Boolean> {
```

yaPOSH

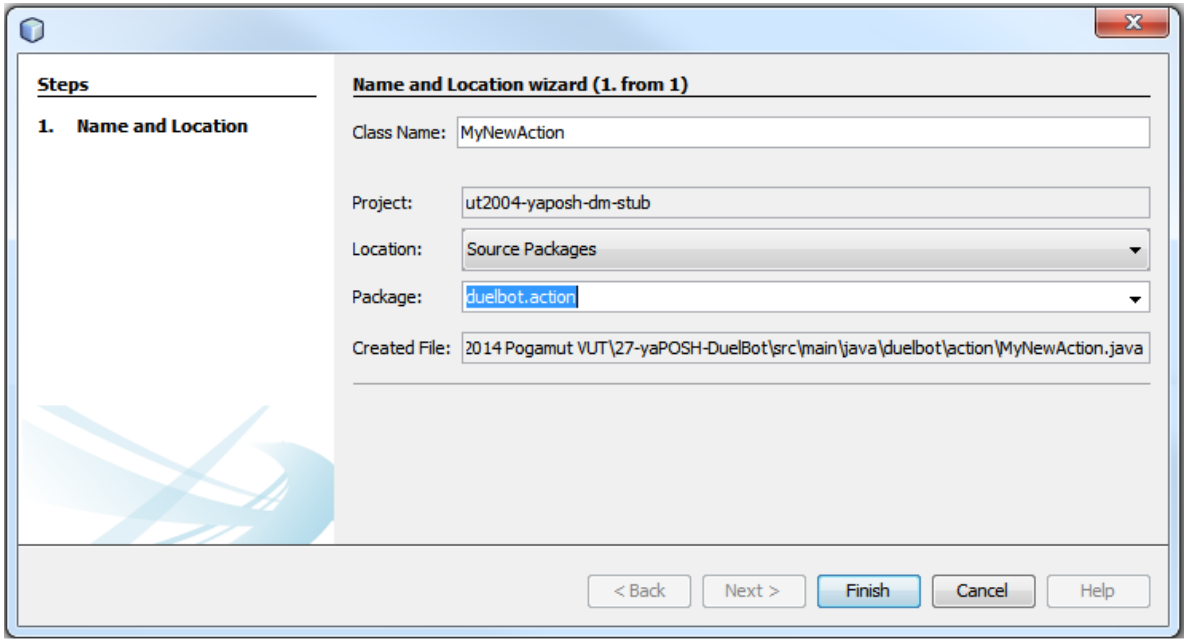
How to make new Action?

1.) DRAG & DROP!





2.) Fill template

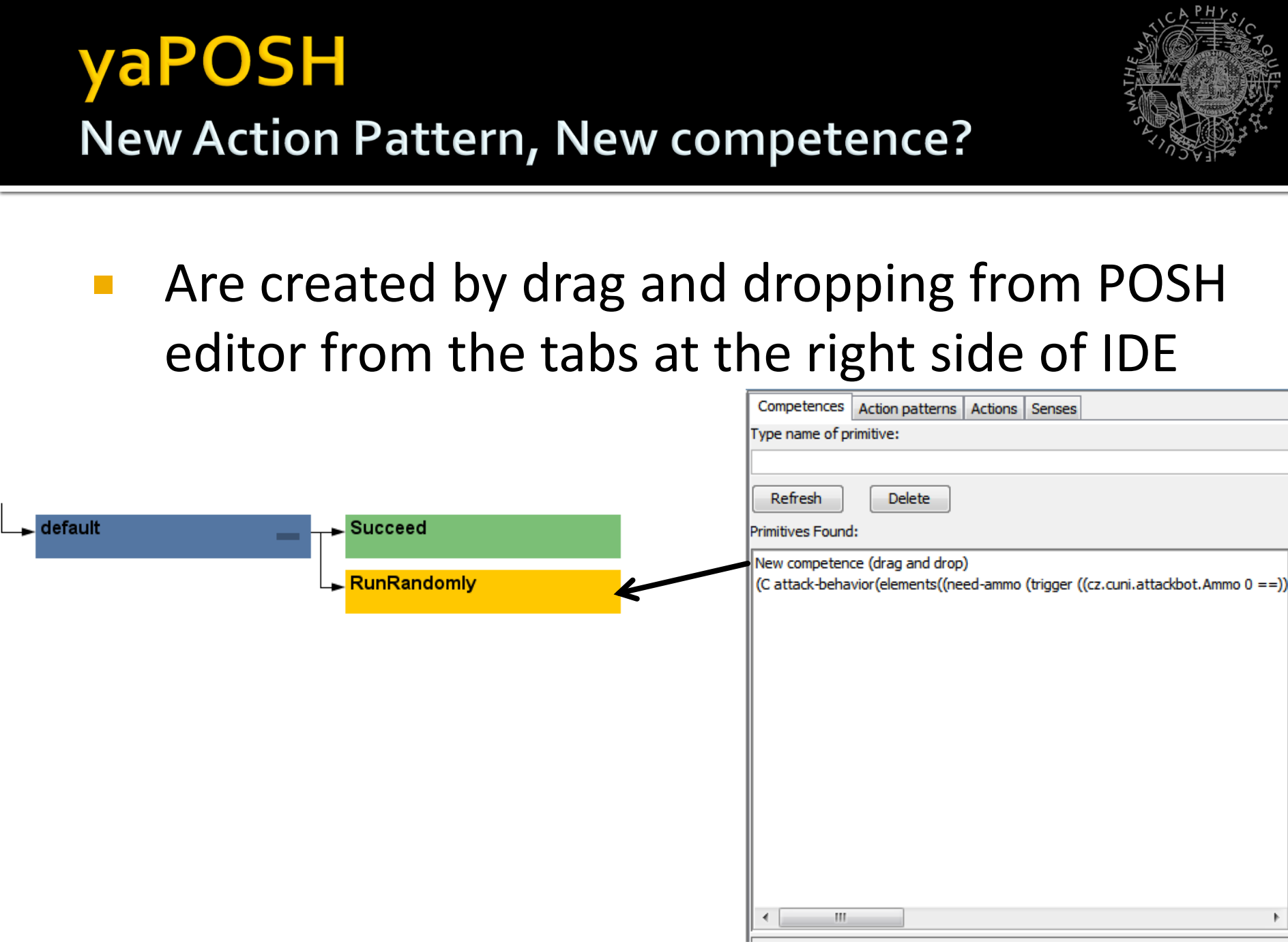


3.) Edit generated Java source file

```
@PrimitiveInfo(name="Shoot the player", description="Shoot the player.")
public class ShootPlayer extends ParamsAction<AttackBotContext> {
```

yaPOSH

New Action Pattern, New competence?



- Are created by drag and dropping from POSH editor from the tabs at the right side of IDE

yaPOSH Context

How to access Pogamut modules?

- Every POSH action and sense has *Context* (this.ctx) that contains all Pogamut modules.
- *Context* is an editable class that is a part of your POSH bot sources, e.g. **AttackBotContext**
- You may use context to store some variables, e.g. *Item* you are going for or *Player* you are going to fight

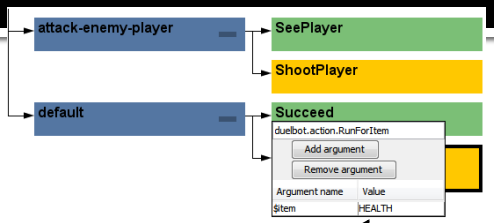
yaPOSH
Parameters

- Competences, action patterns, actions and senses can be parameterized

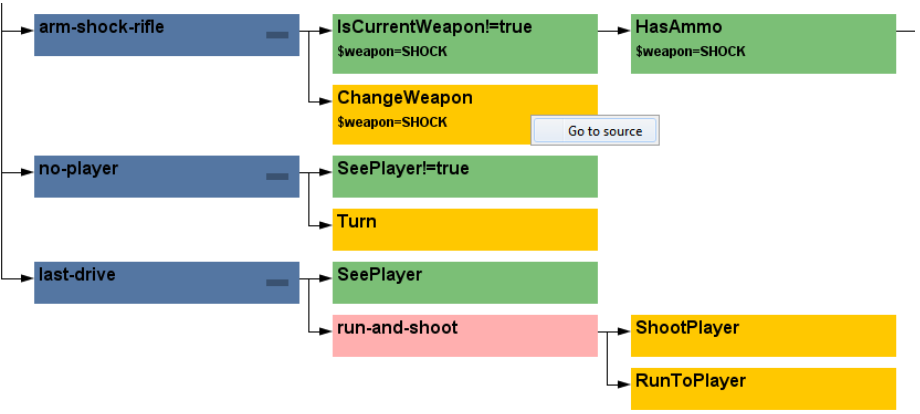
```
(
 (AP go-to-flag
  vars($target="enemy")
   (bot.TurnToFlag($teamname=$target)
    bot.GoToFlag($team=$target)
   )
 )

 (DC life
  (drives
   (
    (pickup-our-flag
     (trigger
      (
       (bot.FlagState($teamname="our")
                      "dropped")
       (bot.FlagIsVisible($teamname="our"))
     ))
     go-to-flag($target="our")
    )
   )
  )
 )
)
```

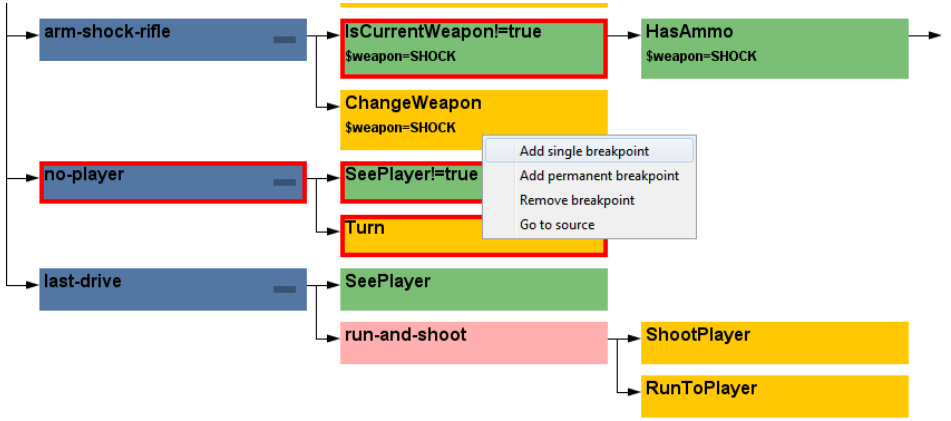
```
@PrimitiveInfo(name        = "Is flag visible",
               description = "our / enemy")
public class FlagVisible
       extends FlagSense<AttackBotContext,Boolean>
{
    public Boolean query(
           @Param("$teamname") String teamname
    ) {
        FlagInfo flag = getFlagInfo(teamname);
        return flag.isVisible();
    }

@PrimitiveInfo(name        = "Turn to flag",
               description = "our / enemy")
public class TurnToFlag
       extends FlagAction<AttackBotContext> {

    public ActionResult run(
           @Param("$teamname") String teamName
    ) {
        FlagInfo flag = getFlagInfo(teamName);
        ctx.getMove().turnTo(flag.getLocation());
        return ActionResult.RUNNING_ONCE;
    }
```

yaPOSH
POSH Editor



- Enables drag and drop
  - Select action or sense you want to add or change from the editor and drag and drop it at desired place
- Double clicking POSH graphical element opens editor, right clicking opens element menu
- Support “Go to source”, breakpoints and debugging
- Breakpoints **PAUSE** the **bot <u>AND</u>** the **environment**



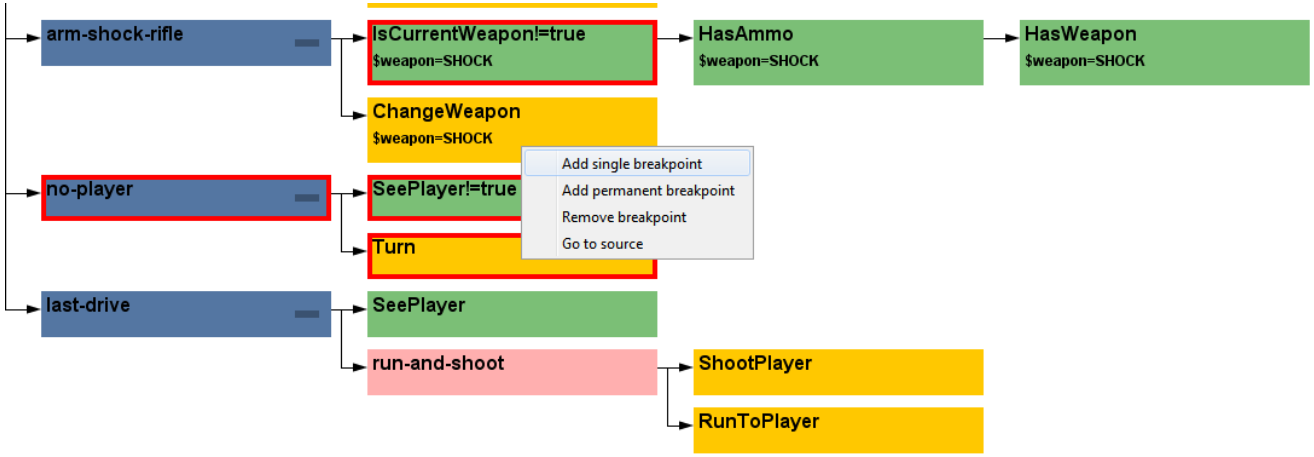


yaPOSH

How to run POSH plan debugger

- Run the bot in **Debug mode** (right click the project, select **Debug**)
- In the Debug toolbar, click the *green circle* button to enable POSH plan debugger

- A window with Debugger appears:



Practice Lesson
Outline

1. Big Picture
2. BOD & POSH
3. yaPOSH
4. **Simple DMBot in yaPOSH**

Assignment 9
(or Homework)

- Create (Simple) **DeathMatchBot** in yaPOSH
  - That arms himself and is able to fight an opponent
  - Does not stuck (for long).

- Points: 5

Assignment 9
Cheatsheet

- Access Pogamut modules from POSH actions and senses!
  - `this.ctx.getItems().getSpawnedItems(UT2004ItemType.Category.WEAPON)`
  - `MyCollections.getFiltered(Collection, new IFilter<Item>() {…})`
- Handling unreachable items:
  - `this.ctx.getNavigation().addStrongNavigationListener(…STUCK_EVENT…)`
  - `myTabooSet.add() & myTabooSet.filter(…)`
- Specifying weapon preferences:
  - `this.ctx.getWeaponPrefs().addGeneralPref(UT2004ItemType.FLAK_CANNON,true)`
    `.addGeneralPref(UT2004ItemType.ROCKET_LAUNCHER,true);`

Send us finished assignment

Via e-mail:

- *Subject*
  - "Pogamut homework 2015 – Assignment X"
    - Replace `'X'` with the assignment number and the subject has to be without quotes of course
    - …or face -2 score penalization
- *To*
  - jakub.gemrot@gmail.com
    - Jakub Gemrot (Tuesday practice lessons)
- *Attachment*
  - Completely zip-up your project(s) folder except `'target'` directory and `IDE specific files` (or face -2 score penalization)
- *Body*
  - **Please send us information about how much time it took you to finish the assignment + any comments regarding your implementation struggle**
    - *Information won't be abused/made public*
    - *In fact it helps to make the practice lessons better*
  - Don't forget to mention your full name!

Questions?

I sense a soul in search of answers…

- We do not own the patent of perfection (yet…)
- In case of doubts about the assignment, tournament or hard problems, bugs don't hesitate to contact us!
  - Jakub Gemrot (Tuesday practice lessons)
    - jakub.gemrot@gmail.com