VGP332 – Artificial Intelligence

Instructor: Peter Chan

Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview

Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview

Assignment 6

- Questions?

Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview

Goal-Driven Behaviour

- Discuss typical day example

Goal-Driven Behaviour

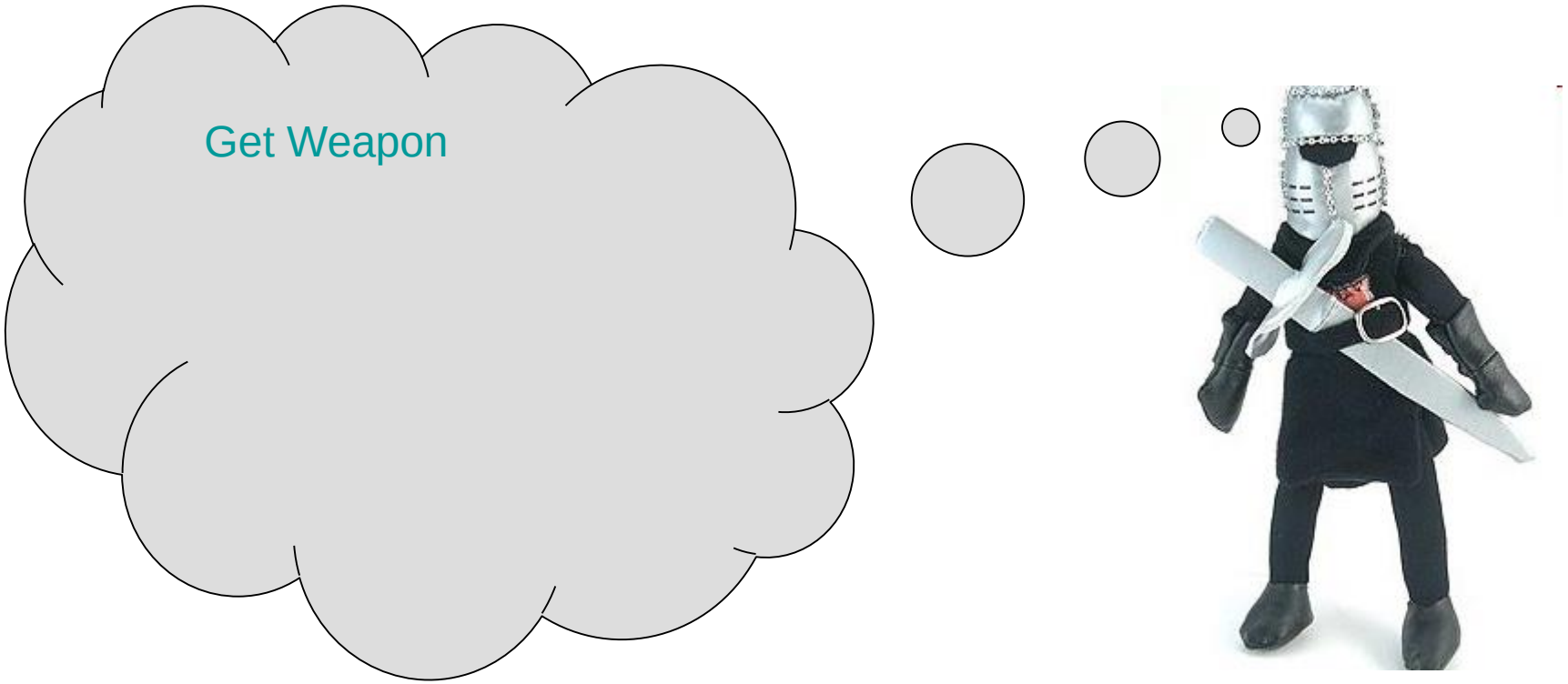
- Hierarchical
- Atomic or composite
- Mimics human thought processes

Goal-Driven Behaviour Example

- Strategic Goal: “Get Weapon”

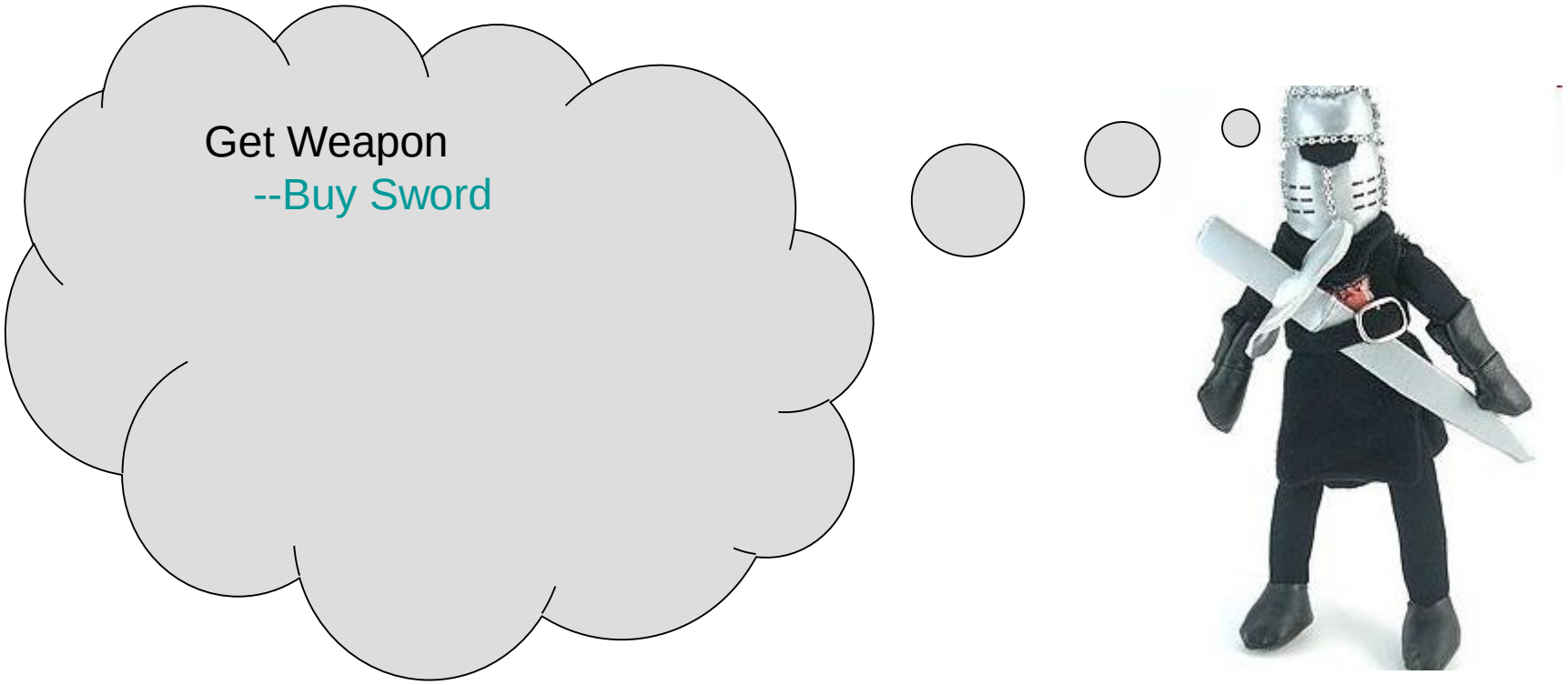
Goal-Driven Behaviour Example

- Strategic Goal: “Get Weapon”



Goal-Driven Behaviour Example

- Decompose into “Buy Sword”



Goal-Driven Behaviour Example

- “Buy Sword” still a composite goal
- Decompose into sub goals

Get Weapon
--Buy Sword
--Get Gold
--Go to Blacksmith

Goal-Driven Behaviour Example

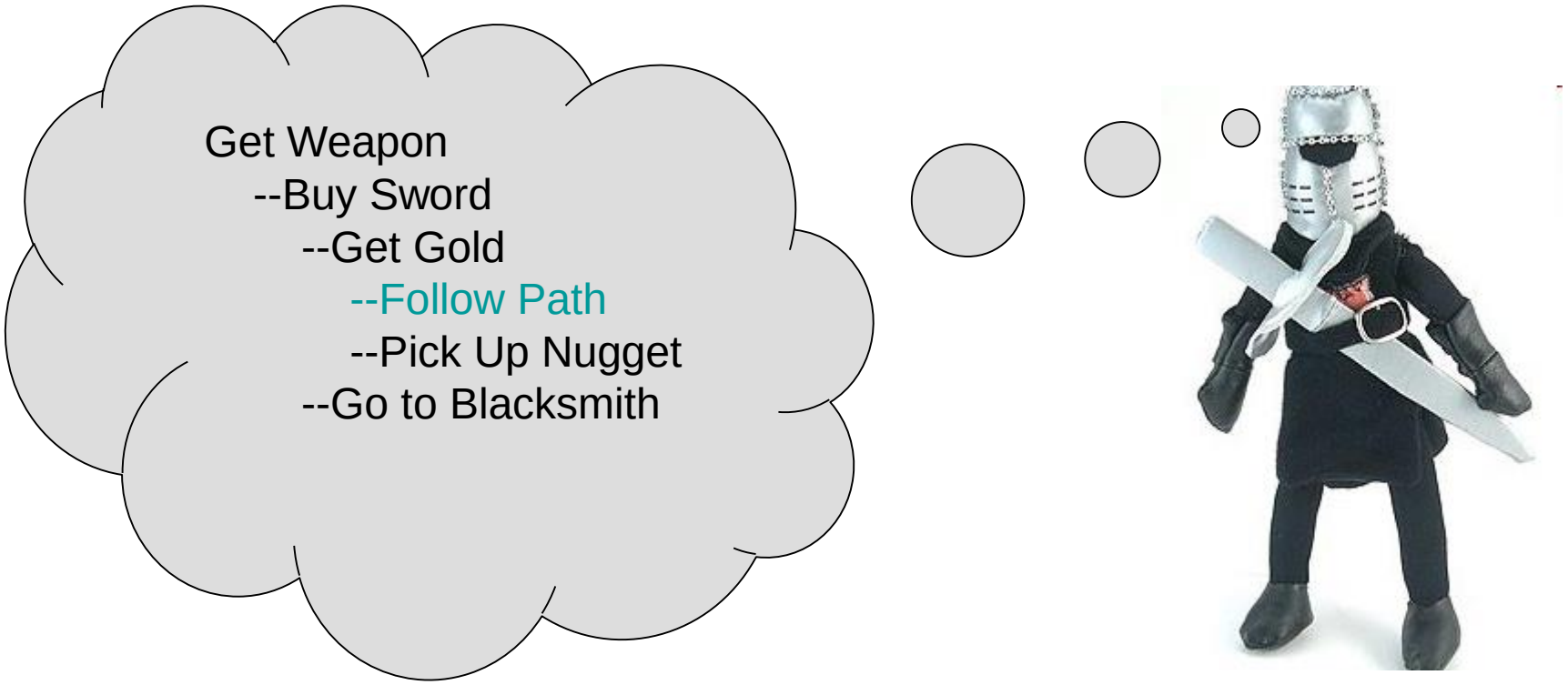
- "Get Gold" needs to be decomposed too

Get Weapon
--Buy Sword
--Get Gold
--Plan Path (Mine)
--Follow Path
--Pick Up Nugget
--Go to Blacksmith

"Go to Blacksmith" goal not expanded yet because not reached

Goal-Driven Behaviour Example

- “Plan Path” returns a list of waypoints to Mine
- Once goal satisfied, it is removed from list



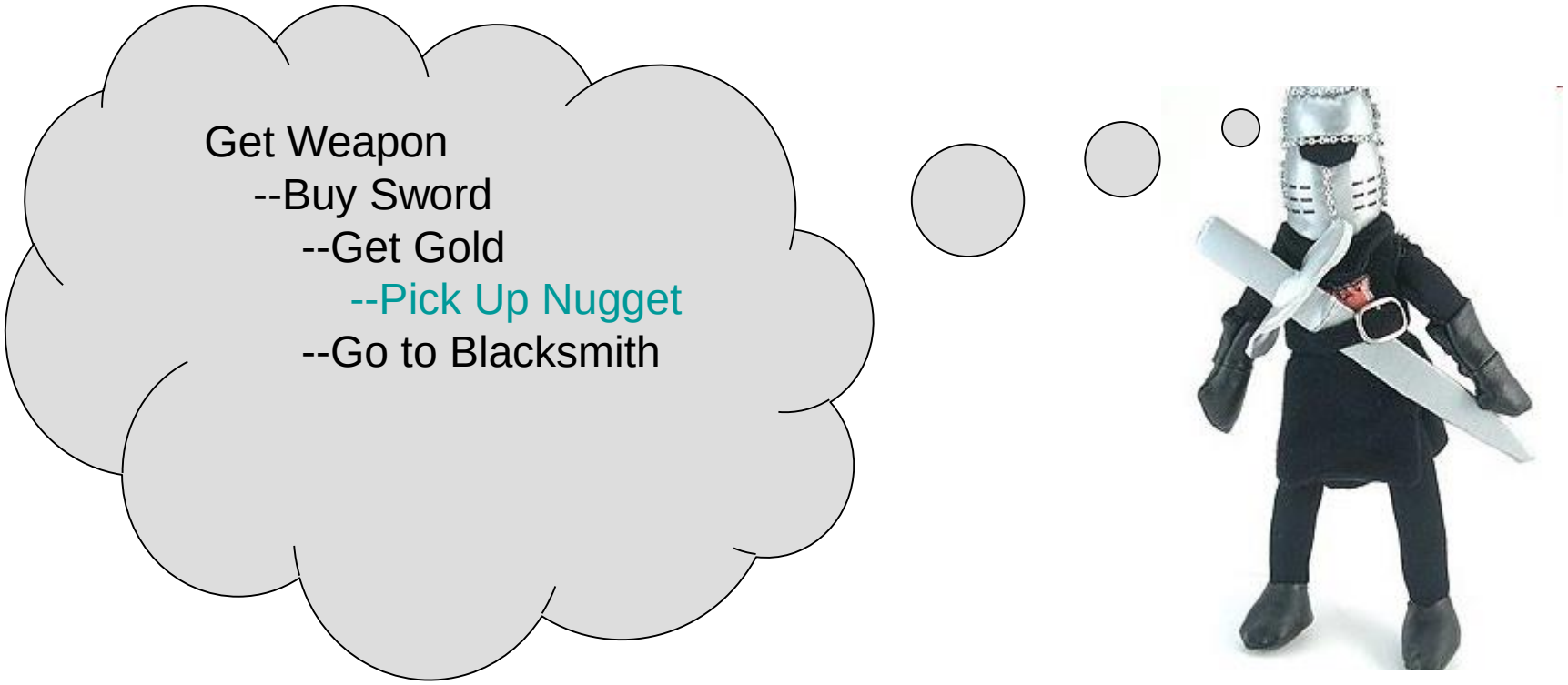
Goal-Driven Behaviour Example

- “Follow Path” expanded appropriately

Get Weapon
--Buy Sword
--Get Gold
--Follow Path
--Traverse Edge (1)
--Traverse Edge (2)
--Traverse Edge (3)
--Pick Up Nugget
--Go to Blacksmith

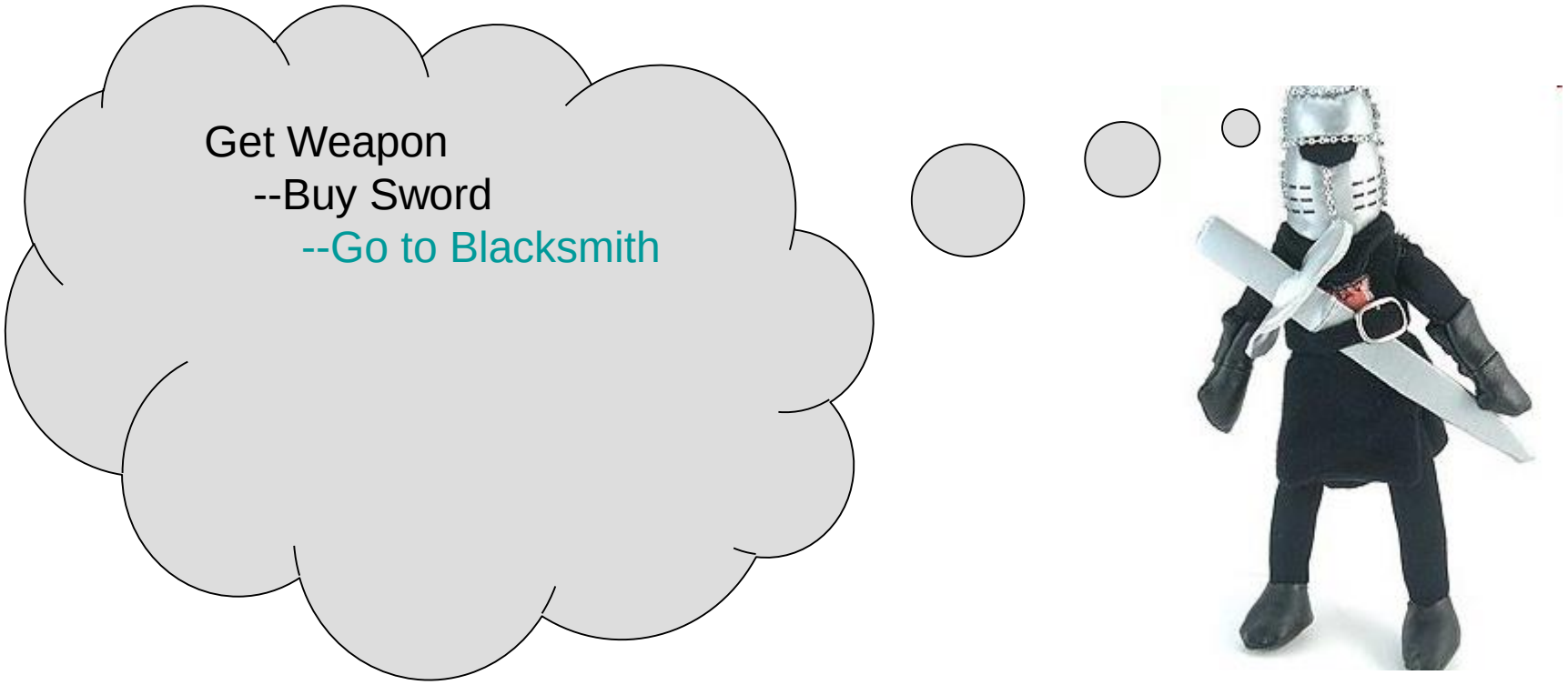
Goal-Driven Behaviour Example

- Once all edges traversed, “Follow Path” satisfied



Goal-Driven Behaviour Example

- When nugget picked up, “Get Gold” satisfied
- “Go to Blacksmith” now needs to be expanded



Goal-Driven Behaviour Example

- When nugget picked up, “Get Gold” satisfied
- “Go to Blacksmith” now needs to be expanded

Get Weapon
--Buy Sword
--Go to Blacksmith
--Plan Path (Smith)
--Follow Path
--Buy Sword

Goal-Driven Behaviour Example

- “Plan Path” returns waypoints to Blacksmith
- “Follow Path” decomposed into edge traversal

Get Weapon
--Buy Sword
--Go to Blacksmith
--Follow Path
--Traverse Edge (1)
--Traverse Edge (2)
--Traverse Edge (3)
--Buy Sword

Goal-Driven Behaviour Example

- At the Blacksmith, pay for sword with nugget

Get Weapon
--Buy Sword
--Go to Blacksmith
--Buy Sword

Goal-Driven Behaviour Example

- All subgoals satisfied
- Strategic top-level goal satisfied

Goal-Driven Behaviour Example

- Try out a different example

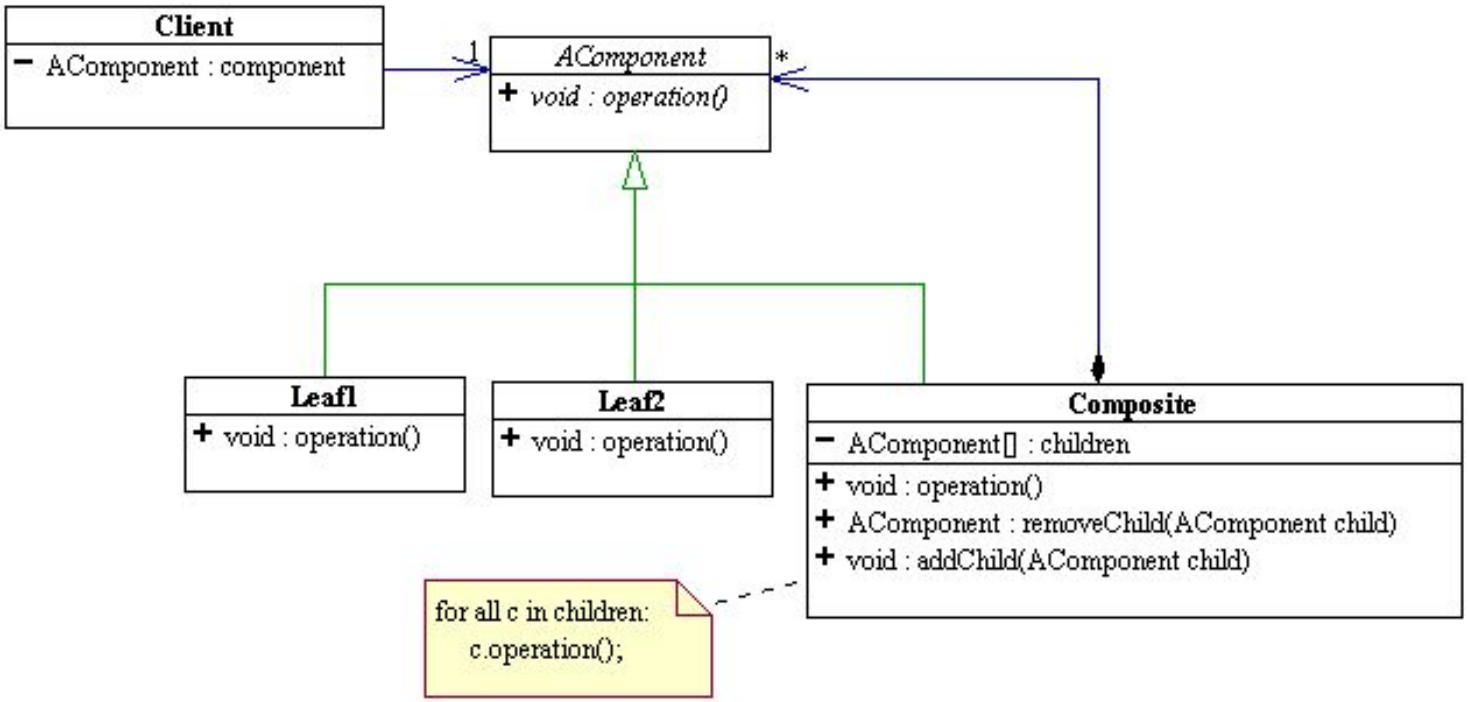
Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview

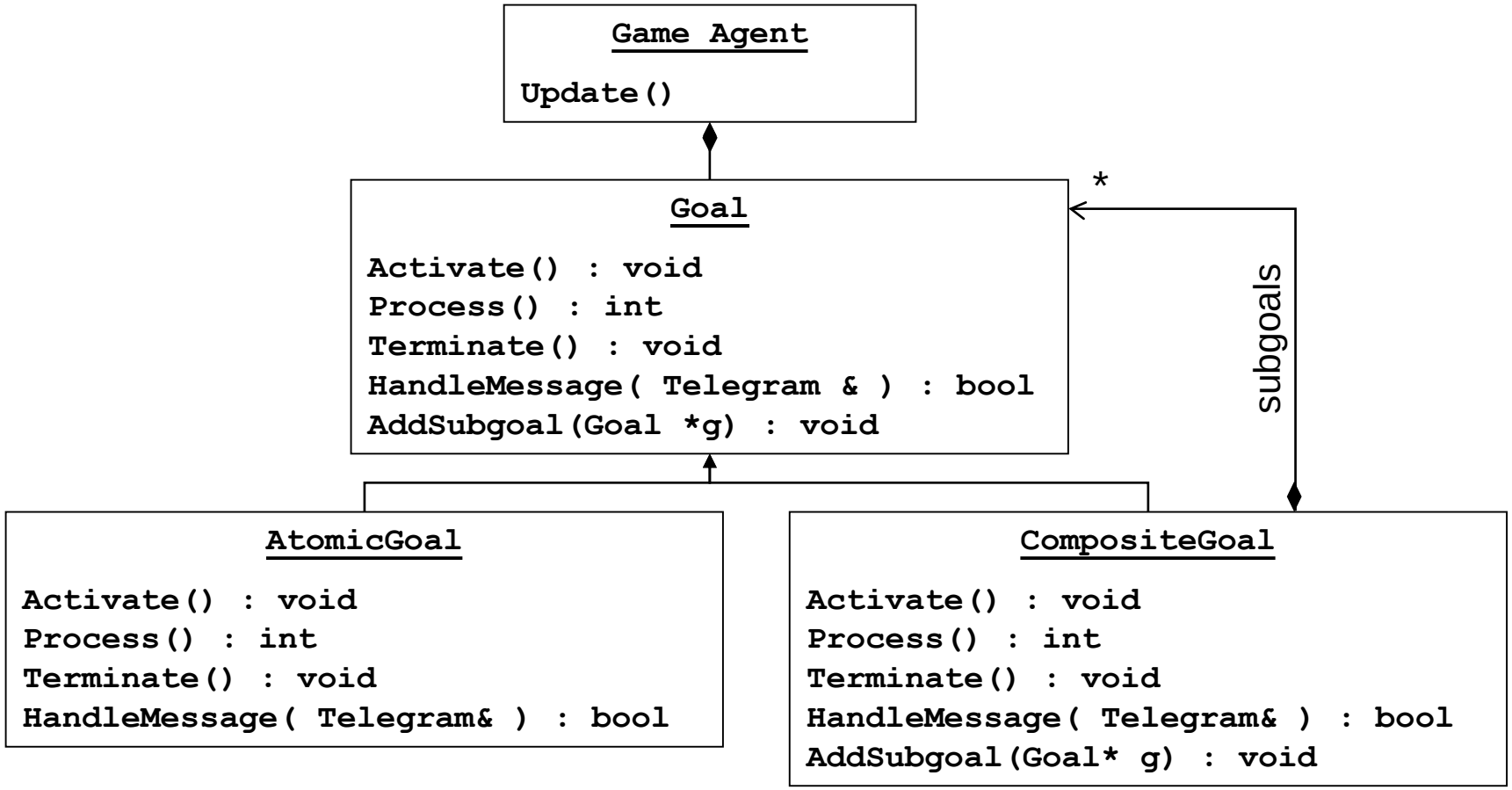
Goal Architecture

- Should handle both atomic and composite goals
- Use composite design pattern
  - **`Component`** can be **`Atomic`** or **`Composite`**
  - **`Composite`** objects are collections of **`Component`**s
  - Requests to **`Composite`** forwarded to all children

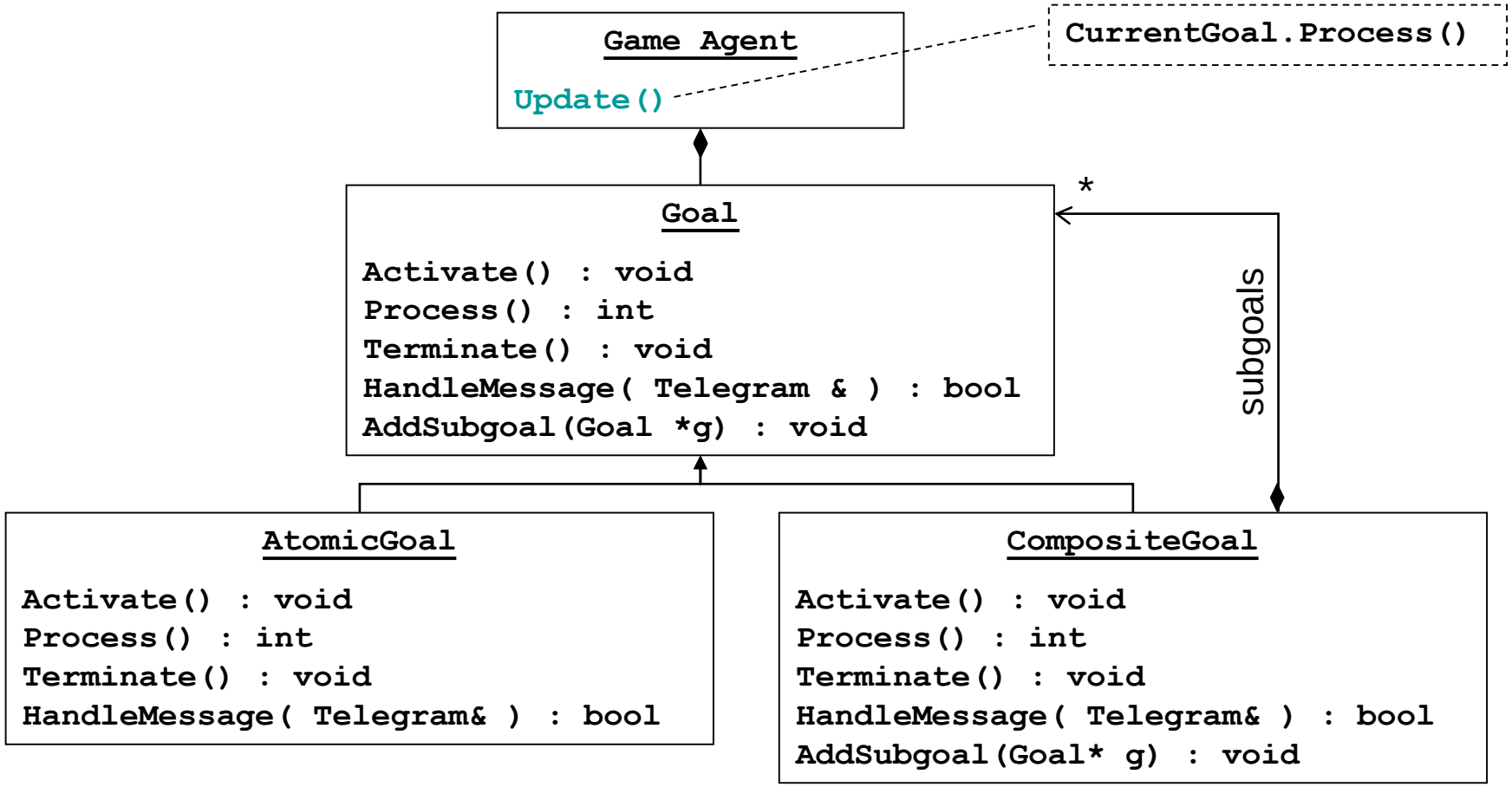
Composite Design Pattern



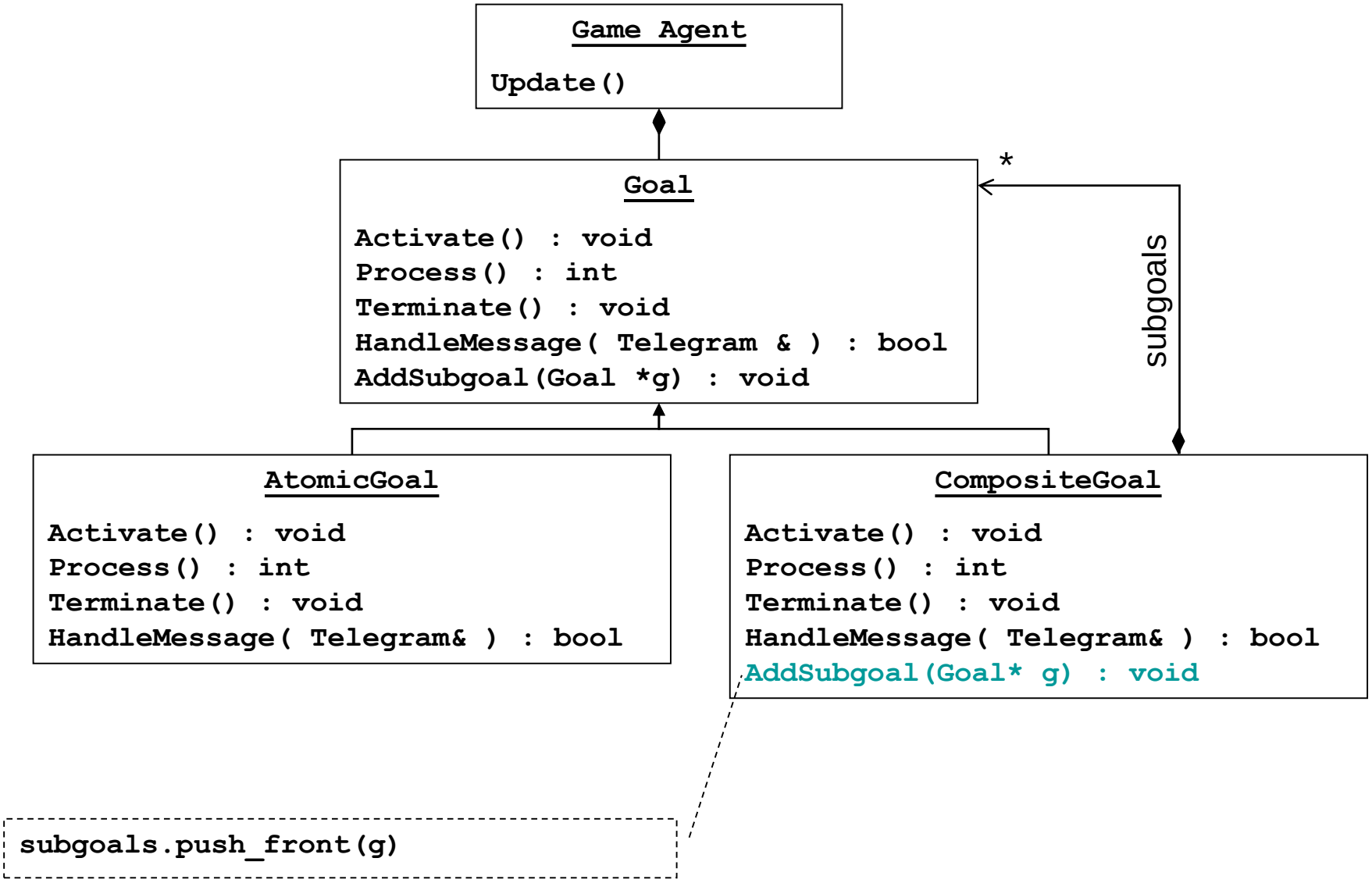
Goal Architecture



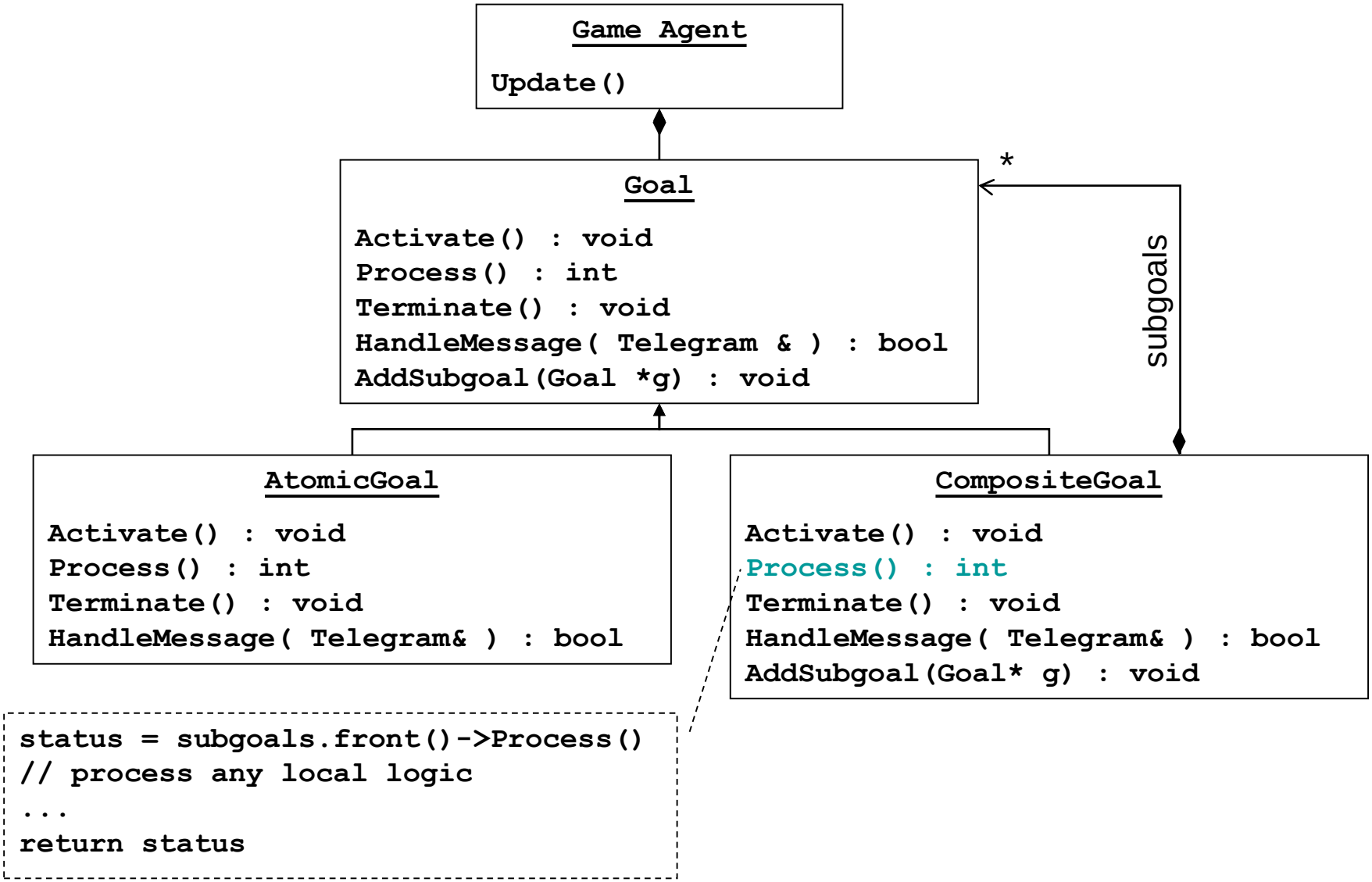
Goal Architecture



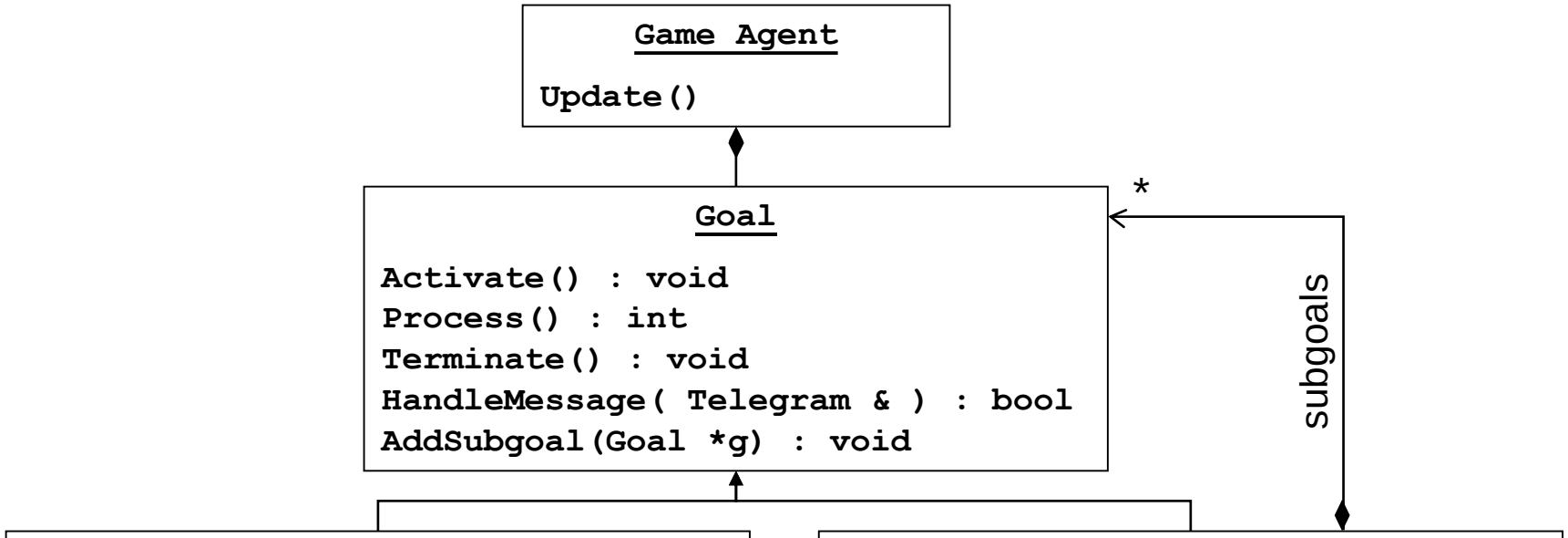
Goal Architecture



Goal Architecture



Goal Architecture




```
bool bResult =
  subgoals.front()->HandleMessage(msg)
if (bResult == false)
   // attempt to handle message locally
```

Goal Architecture

- **`Activate`**
- **`Process`**
- **`Terminate`**
- **`HandleMessage`**

Goal Architecture

- **`Activate`** → **`Enter`**
- **`Process`** → **`Execute`**
- **`Terminate`** → **`Exit`**
- **`HandleMessage`** → **`HandleMessage`**

Similar to **`State`** class:

Goal Architecture

- **`Activate`**
- **`Process`**
- **`Terminate`**
- **`HandleMessage`**

- Initialization logic
- Represents planning phase of goal
- May be called multiple times to replan goal if situation changes

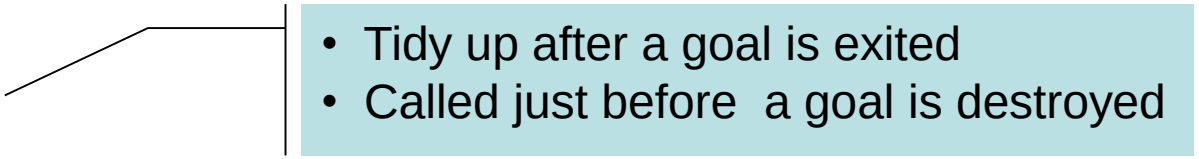
Goal Architecture

- **`Activate`**
- **`Process`**
- **`Terminate`**
- **`HandleMessage`**

- Executed once per update step
- Returns enum value:
  - inactive : goal is waiting for activation
  - active : goal will be processed each update
  - completed : goal finished, will be removed next update
  - failed : goal either needs replanning or will be removed next update

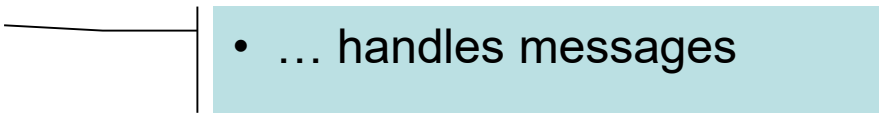
Goal Architecture

- **`Activate`**
- **`Process`**
- **`Terminate`**
- **`HandleMessage`**



Goal Architecture

- **`Activate`**
- **`Process`**
- **`Terminate`**
- **`HandleMessage`**



Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview

Raven Bots Goals

Composite Goals

Goal_Think
Goal_GetItem
Goal_MoveToPosition
Goal_FollowPath
Goal_Explore
Goal_AttackTarget
Goal_HuntTarget

Atomic Goals

Goal_Wander
Goal_SeekToPosition
Goal_TraverseEdge
Goal_DodgeSideToSide

Raven Bots Goals

Composite Goals

Goal_Think
Goal_GetItem
Goal_MoveToPosition
Goal_FollowPath
Goal_Explore
Goal_AttackTarget
Goal_HuntTarget

Atomic Goals

Goal_Wander
Goal_SeekToPosition
Goal_TraverseEdge
Goal_DodgeSideToSide

Goal_Wander

- Atomic goal

```
class Goal_Wander : public Goal<Raven_Bot>
{
public:
   Goal_Wander(Raven_Bot* pBot):Goal<Raven_Bot>(pBot, goal_wander){}

   void Activate();
   int Process();
   void Terminate();
};
```

Goal_Wander

- **`Activate`** method

```
void Goal_Wander::Activate()
{
   m_Status = active;

   m_pOwner->GetSteering()->WanderOn();
}
```

Goal_Wander

- **`Process`** method

```
int Goal_Wander::Process()
{
   // If status is inactive, call Activate() and set status to active
   ActivateIfInactive();

   return m_Status;
}
```

Goal_Wander

- **`Terminate`** method

```
void Goal_Wander::Terminate()
{
   m_pOwner->GetSteering()->WanderOff();
}
```

Goal_TraverseEdge

- Atomic goal

```
class Goal_TraverseEdge : public Goal<Raven_Bot>
{
private:
   PathEdge m_Edge;  // the edge the bot will follow
   bool m_bLastEdgeInPath;  // true if m_Edge is the last in the path
   double m_dTimeExpected;  // estimated time bot should take to traverse edge
   double m_dStartTime;  // time this goal was activated
   bool isStuck() const;  // returns true if bot gets stuck

public:
   Goal_TraverseEdge(Raven_Bot* pBot, PathEdge edge, bool LastEdge);

   void Activate();
   int Process();
   void Terminate();
};
```

Goal_TraverseEdge

- **`Activate`** method

```
void Goal_TraverseEdge::Activate()
{
   m_Status = active;

   // edge behavior flag may specify movement type
   switch (m_Edge.GetBehaviorFlag())
   {
      case NavGraphEdge::swim:
         m_pOwner->SetMaxSpeed(script->GetDouble("Bot_MaxSwimmingSpeed"));
         break;
      case NavGraphEdge::crawl:
         m_pOwner->SetMaxSpeed(script->GetDouble("Bot_MaxCrawlingSpeed"));
         break;
   }

   // record time bot starts this goal
   m_dStartTime = Clock->GetCurrentTime();
```

…contd.

Goal_TraverseEdge

- **`Activate`** method

```
   // calculate expected time required to reach this waypoint
   m_dTimeExpected =
      m_pOwner->CalculateTimeToReachPosition(m_Edge.GetDestination());

   // factor in a margin of error
   static const double MarginOfError = 2.0;
   m_dTimeExpected += MarginOfError;

   // set steering target
   m_pOwner->GetSteering()->SetTarget(m_Edge.GetDestination());

   // if this is the last edge, bot needs to arrive; else seek
   if (m_bLastEdgeInPath)
      m_pOwner->GetSteering()->ArriveOn();
   else
      m_pOwner->GetSteering()->SeekOn();
}
```

Goal_TraverseEdge

- **`Process`** method

```
int Goal_TraverseEdge::Process()
{
   // if status is inactive, call Activate()
   ActivateIfInactive();

   // if bot has become stuck, return failure
   if (isStuck())
      m_Status = failed;

   // if bot has reached end of edge, return completed
   else
   {
      if (m_pOwner->isAtPosition(m_Edge.GetDestination())
         m_Status = completed;
   }

   return m_Status;
}
```

Goal_TraverseEdge

- **`Terminate`** method

```
void Goal_TraverseEdge::Terminate()
{
   // turn off steering behaviors
   m_pOwner->GetSteering()->SeekOff();
   m_pOwner->GetSteering()->ArriveOff();

   // return max speed back to normal
   m_pOwner->SetMaxSpeed(script->GetDouble("Bot_MaxSpeed"));
}
```

Goal_FollowPath

- Composite goal

```
class Goal_FollowPath : public Goal_Composite<Raven_Bot>
{
private:
   // a local copy of the path returned by path planner
   std::list<PathEdge> m_Path;

public:
   Goal_FollowPath(Raven_Bot* pBot, std::list<PathEdge> path);

   void Activate();
   int Process();
   void Terminate();
};
```

Goal_FollowPath

- **`Activate`** method

```
void Goal_FollowPath::Activate()
{
   m_iStatus = active;
   PathEdge edge = m_Path.front();   // get reference to next edge
   m_Path.pop_front();   // remove edge from path
   // add appropriate navigation subgoal based on edge type
   switch (edge.GetBehaviorFlags())
   {
      case NavGraphEdge::normal:
         { AddSubgoal(new Goal_TraverseEdge(m_pOwner, edge, m_Path.empty())); }
         break;
      case NavGraphEdge::goes_through_door:
         { AddSubgoal(new Goal_NegotiateDoor(m_pOwner, edge, m_Path.empty())); }
         break;
      case NavGraphEdge::jump: /* add subgoal to jump along edge */ break;
      case NavGraphEdge::grapple: /* add subgoal to grapple along edge */ break;
      default:
         throw std::runtime_error("unrecognized edge type");
   }
}
```

Goal_FollowPath

- **`Process`** method

```
int Goal_FollowPath::Process()
{
   // if status is inactive, call Activate()
   ActivateIfInactive();

   // if no subgoals and there is still edge left to traverse,
   // add edge as a subgoal
   m_Status = ProcessSubgoals();

   if (m_Status == completed && !m_Path.empty())
   {
      Activate();
   }

   return m_Status;
}
```

Goal_MoveToPosition

- Composite goal

```
class Goal_MoveToPosition : public Goal_Composite<Raven_Bot>
{
private:
   // the position the bot wants to reach
   Vector2D m_vDestination;

public:
   Goal_MoveToPosition(Raven_Bot* pBot, Vector2D pos);

   void Activate();
   int Process();
   void Terminate();

   // this goal is able to accept messages
   bool HandleMessage(const Telegram& msg);
};
```

Goal_MoveToPosition

- **`Activate`** method

```
void Goal_MoveToPosition::Activate()
{
   m_Status = active;

   RemoveAllSubgoals();   // make sure subgoal list is clear

   // request path from path planner
   // while time-sliced path planning taking place, seek to position
   if (m_pOwner->GetPathPlanner()->RequestPathToTarget(m_vDestination))
   {
      AddSubgoal(new Goal_SeekToPosition(m_pOwner, m_vDestination));
   }
}
```

Goal_MoveToPosition

- **`HandleMessage`** method

```
bool Goal_MoveToPosition::HandleMessage(const Telegram& msg)
{
   // first, pass message down the goal hierarchy
   bool bHandled = ForwardMessageToFrontMostSubgoal(msg);
   // if msg not handled, see if this goal can handle it
   if (!bHandled)
   {
      switch (msg.Msg)
      {
         case Msg_PathReady:
            RemoveAllSubgoals();   // clear any existing subgoals
            AddSubgoal(new Goal_FollowPath(m_pOwner,
                       m_pOwner->GetPathPlanner()->GetPath()));
            return true;   // msg handled
         case Msg_NoPathAvailable:
            m_Status = failed;
            return true;   // msg handled
         default: return false;
      }
   }
   return true;   // handled by subgoals
}
```

Goal_MoveToPosition

- **`Process`** method

```
int Goal_MoveToPosition::Process()
{
   // if status is inactive, call Activate()
   ActivateIfInactive();

   m_Status = ProcessSubgoals();   // process subgoals

   // if any subgoals fail, this goal needs to replan
   ReactivateIfFailed();

   return m_Status;
}
```

Goal_AttackTarget

- Composite goal

```
class Goal_AttackTarget : public Goal_Composite<Raven_Bot>
{
public:
   Goal_AttackTarget(Raven_Bot* pOwner);

   void Activate();
   int Process();
   void Terminate() { m_iStatus = completed; }
};
```

Goal_AttackTarget

- **`Activate`** method

```
void Goal_AttackTarget::Activate()
{
   m_iStatus = active;

   // if this goal is reactivated, pre-existing subgoals must be removed
   RemoveAllSubgoals();

   // bot's target may die while this goal is still active
   // so test to ensure that bot always has a viable target
   if (!m_pOwner->GetTargetSys()->isTargetPresent())
   {
      m_iStatus = completed;
      return;
   }
```

…contd.

Goal_AttackTarget

- **`Activate`** method

```
   // if bot able to shoot target, then select appropriate tactic
   if (m_pOnwer->GetTargetSys()->isTargetShootable())
   {
      // strafing possible?
      Vector2D dummy;
      if (m_pOwner->canStepLeft(dummy) || m_pOwner->canStepRight(dummy))
         AddSubgoal(new Goal_DodgeSideToSide(m_pOwner));
      // otherwise head directly to target
      else
         AddSubgoal(new Goal_SeekToPosition(m_pOwner,
                     m_pOwner->GetTargetBot()->Pos()));
   }
   // target isn't visible, go hunt it
   else
   {
      AddSubgoal(new Goal_HuntTarget(m_pOwner));
   }
}
```

Goal_AttackTarget

- **`Process`** method

```
int Goal_AttackTarget::Process()
{
   // if status inactive, call Activate()
   ActivateIfInactive();

   // process subgoals
   m_iStatus = ProcessSubgoals();

   ReactivateIfFailed();

   return m_iStatus;
}
```

Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview

Goal Arbitration

- **`Goal_Think`**
- Composite goal
- Highest level
- Instantiated once when agent created
- Terminates only when agent destroyed
- Chooses between strategic composite goals

Goal Arbitration

- **`Goal_Think`**
- Composite goal
- Highest level
- Instantiated once when agent created
- Terminates only when agent destroyed
- Chooses between strategic composite goals

Notice any similarity with something we've covered before?

Strategic Goals

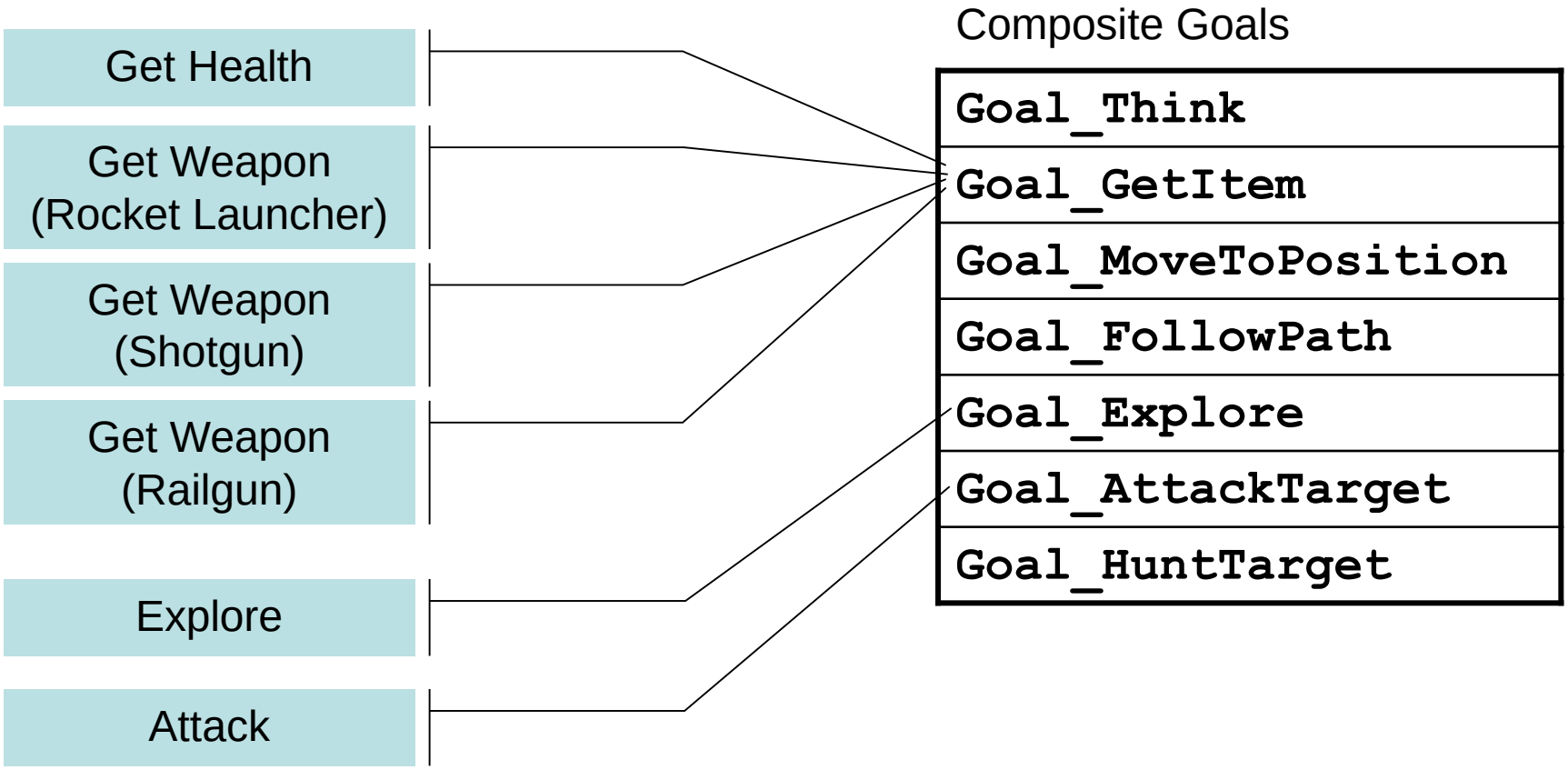
- Recall Raven Bots' composite goals:

Composite Goals

Goal_Think
Goal_GetItem
Goal_MoveToPosition
Goal_FollowPath
Goal_Explore
Goal_AttackTarget
Goal_HuntTarget

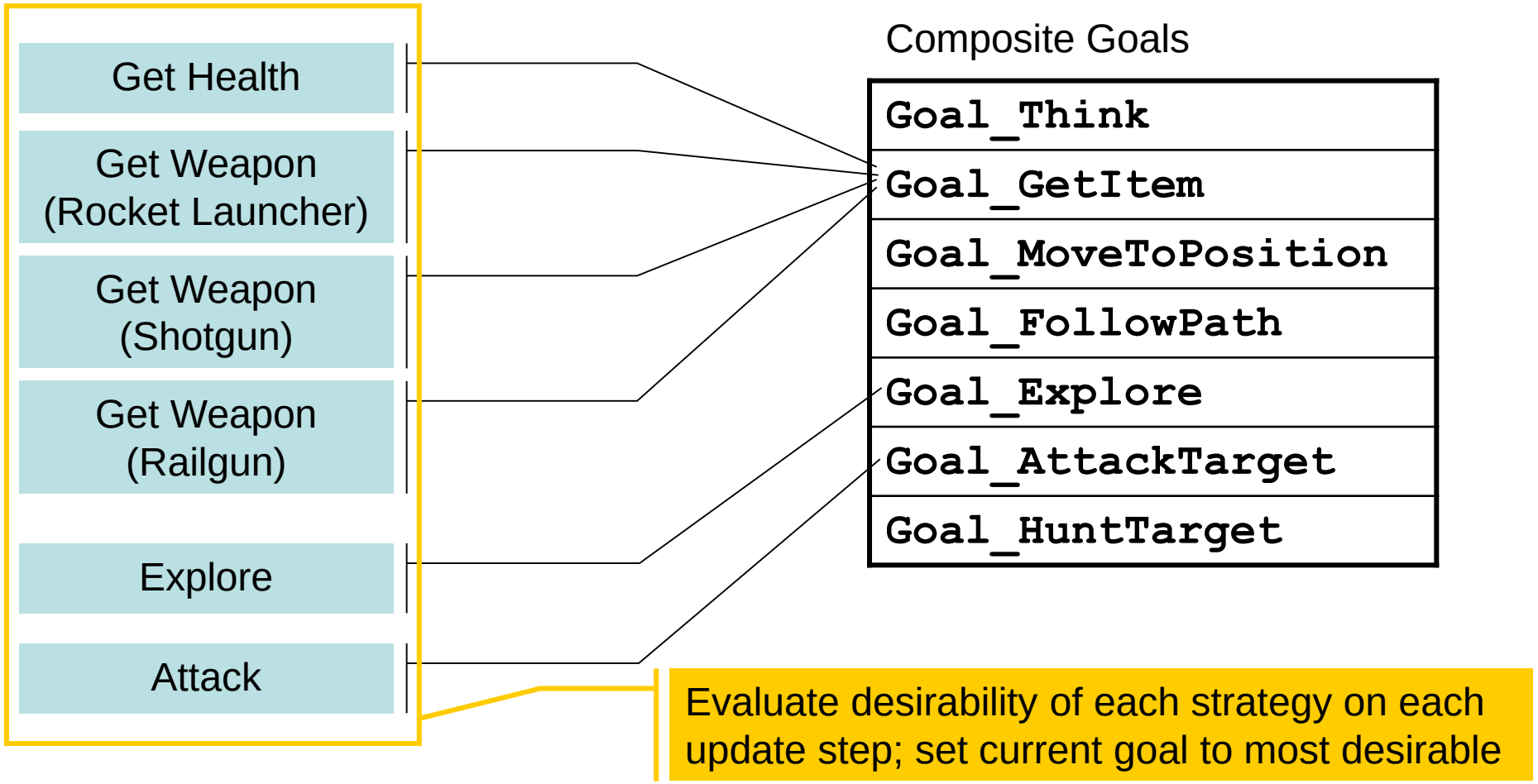
Strategic Goals

- Create Raven Bot strategy:



Strategic Goals

- **`Goal_Think`** picks a strategy



Strategic Goals

- Each strategic goal has a desirability
- Standardize across all goals (e.g., 0 to 1)
- Common ratings can be encapsulated

Strategic Goals

- Each strategic goal has a desirability
- Standardize across all goals (e.g., 0 to 1)
- Common ratings can be encapsulated

```
class Raven_Feature
{
public:
   // Each of the following methods returns a value between 0 and 1
   static double Health(Raven_Bot* pBot);
   static double DistanceToItem(Raven_Bot* pBot, int ItemType);
   static double IndividualWeaponStrength(Raven_Bot* pBot, int WeaponType);
   static double TotalWeaponStrength(Raven_Bot* pBot);
};
```

Health Desirability

- Proportional to how injured a bot is
- Inversely proportional to distance to health pack

$$Desirability_{health} = k \times \frac{1 - Health}{DistToHealth}$$

Tweaking constant

Weapon Desirability

- Proportional to how healthy a bot is
- Proportional to bot's ammo depletion
- Inversely proportional to distance to weapon

$$Desirability_{weapon} = k \times \frac{Health \times (1 - WeaponStrength)}{DistToWeapon}$$

Tweaking constant

Aggro Desirability

- Proportional to bot's total weapon strength
- Proportional to how healthy a bot is

$$Desirability_{attack} = k \times TotalWeaponStrength \times Health$$

Tweaking constant

Exploration Desirability

- When all else fails….

$Desirability_{explore} = 0.05$

Goal Arbitration

- Iterate over strategic goals
- Compute desirability for each
- Activate most desirable strategic goal

Goal Arbitration

- Iterate over strategic goals
- Compute desirability for each
- Activate most desirable strategic goal

```
void Goal_Think::Arbitrate()
{
   double best = 0.0;
   Goal_Evaluator* MostDesirable = NULL;
   // iterate through evaluators to find one with highest score
   GoalEvaluators::iterator curDes = m_Evaluators.begin();
   for (curDes; curDes != m_Evaluators.end(); ++curDes)
   {
      double desirability = (*curDes)->CalculateDesirability(m_pOwner);
      if (desirability >= best)
      {
         best = desirability;
         MostDesirable = *curDes;
      }
   }
   MostDesirable->SetGoal(m_pOwner);
}
```

Putting It All Together

- Try out the Raven demo

Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview

Additional Features

- Goal-driven behaviour lends itself well to many extensions

Agent Personalities

- Each agent has bias for various strategies
- E.g., aggressive, passive, conservative, etc.
- Multiply character bias by goal desirability

Agent Personalities

- Recall Raven bot strategies:
  - Get Health
  - Get Weapon (Rocket Launcher)
  - Get Weapon (Shotgun)
  - Get Weapon (Railgun)
  - Attack
  - Explore

- What biases would you use for:
  - Bot that’s suicidally aggressive
  - Bot that’s super cautious

Anticipation

- Agent can run **`Goal_Think`** for opponent
- Returns a chosen strategy for opponent
- Use this info to anticipate opponent’s actions

State Memory

- Stack nature of goal list allows:
  - Interrupt
  - Resume

State Memory

Get Weapon
--Buy Sword
--Go to Blacksmith
--Follow Path
--Traverse Edge (1)
--Traverse Edge (2)
--Traverse Edge (3)
--Buy Sword

State Memory

Get Weapon
--Buy Sword
--Go to Blacksmith
--Follow Path
--Traverse Edge (1)
--Traverse Edge (2)
--Traverse Edge (3)
--Buy Sword

Have at you!

State Memory

Get Weapon
--Buy Sword
--Go to Blacksmith
--Follow Path
--Defend
--Traverse Edge (1)
--Traverse Edge (2)
--Traverse Edge (3)
--Buy Sword

State Memory

Get Weapon
--Buy Sword
--Go to Blacksmith
--Follow Path
--Traverse Edge (1)
--Traverse Edge (2)
--Traverse Edge (3)
--Buy Sword

State Memory

- Stack nature of goal list also allows:
  - Negotiating special path obstacles

- Example: opening doors on map

State Memory

- Recall **`Goal_FollowPath::Activate`** method

```
// ...

   switch (edge.GetBehaviorFlags())
   {
      case NavGraphEdge::normal:
         { AddSubgoal(new Goal_TraverseEdge(m_pOwner, edge, m_Path.empty())); }
         break;
      case NavGraphEdge::goes_through_door:
         { AddSubgoal(new Goal_NegotiateDoor(m_pOwner, edge, m_Path.empty())); }
         break;
      case NavGraphEdge::jump: /* add subgoal to jump along edge */ break;
      case NavGraphEdge::grapple: /* add subgoal to grapple along edge */ break;
      default:
         throw std::runtime_error("unrecognized edge type");
   }

// ...
```

State Memory

- Add a **`Goal_NegotiateDoor`** composite goal

```
void Goal_NegotiateDoor::Activate()
{
   m_iStatus = active;

   // if this goal is active, existing subgoals must be removed
   RemoveAllSubgoals();

   // get position of closest navigable switch
   Vector2D posSw = m_pOwner->GetWorld()->GetPosOfClosestSwitch(m_pOwner->Pos(),
                                                m_PathEdge.GetDoorID());

   // goals added in reverse order because of stack *push*
   // 3. traverse edge that passes through door
   AddSubgoal(new Goal_TraverseEdge(m_pOwner, m_PathEdge);
   // 2. move to beginning of edge that passes through door
   AddSubgoal(new Goal_MoveToPosition(m_pOwner, m_PathEdge);
   // 1. move to location of the switch
   AddSubgoal(new Goal_MoveToPosition(m_pOwner, posSw);
}
```

State Memory

- Try out Raven example

Command Queuing

- Popular in RTS games
- Started off as path waypoint queuing
- Extended to patrol waypoint queuing
- Extended to *any command* queueing
- Reduces micromanagement

Command Queuing

- Popular in RTS games
- Started off as path waypoint queuing
- Extended to patrol waypoint queuing
- Extended to *any command* queueing
- Reduces micromanagement

- How would you add command queuing using the architecture we've seen?

Scriptable Behaviour

- Goal queue can be used to script linear sequences
- Each step in sequence should have appropriate goal
- Can extend framework to allow scripting in other language (e.g., Lua or Python)

Agenda

- Assignment 6 Redux
- Goal-Driven Behaviour
- Goal Architecture
- Raven Bots Goals
- Goal Arbitration
- Additional Features
- Assignment 7 Overview