AI for games

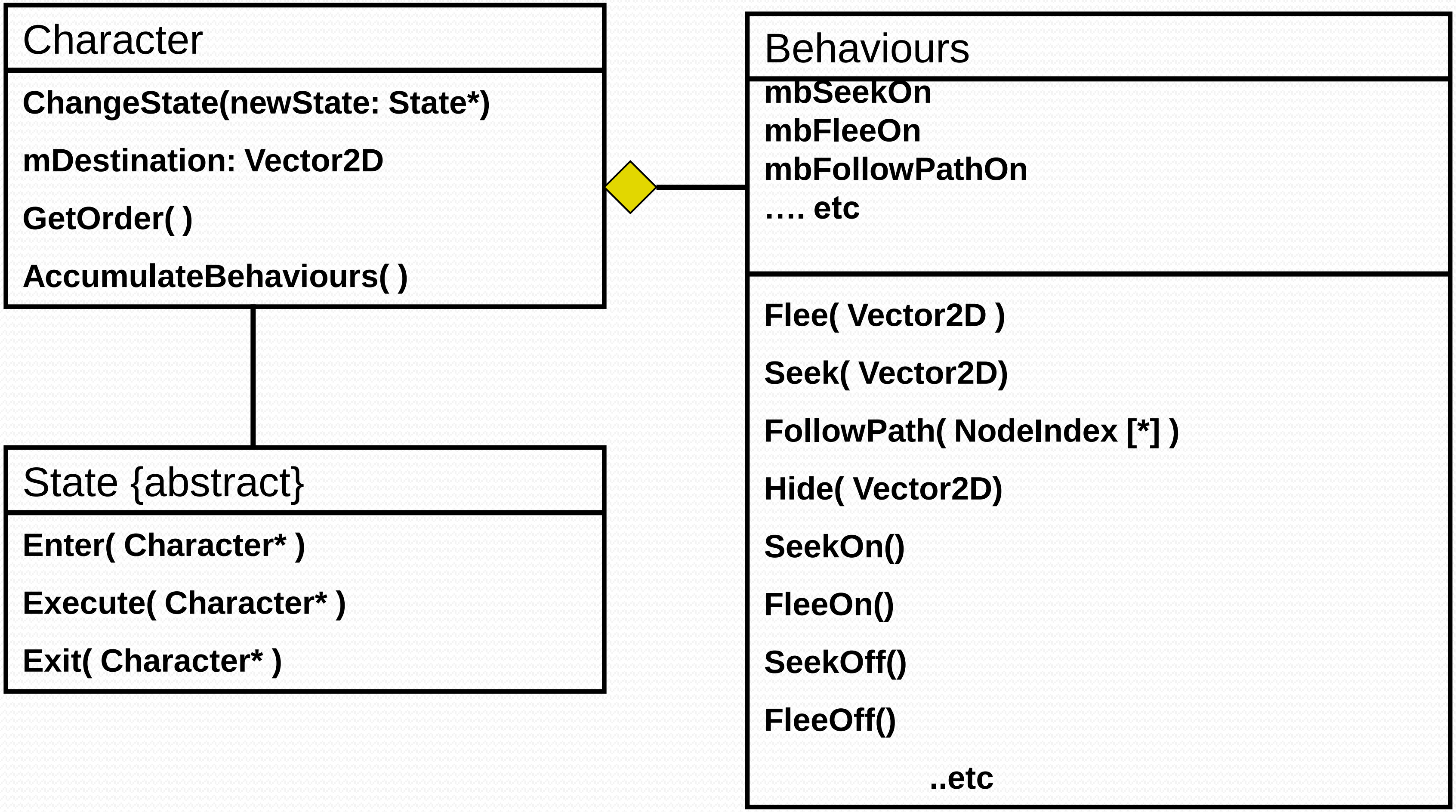
Lecture 7

Applied finite state machines

Organiser

- Quick recap
- Designing the state machine
- Common/emergency state changes
- Tactical evaluation
- Switching superstates
- Passing messages between agents

Reminder



Character

ChangeState(newState: State*)

mDestination: Vector2D

GetOrder( )

AccumulateBehaviours( )

State {abstract}

Enter(Character*)

Execute(Character*)

Exit(Character*)

Behaviours

mbSeekOn
mbFleeOn
mbFollowPathOn
…. etc

Flee(Vector2D)

Seek(Vector2D)

FollowPath(NodeIndex [*])

Hide(Vector2D)

SeekOn()

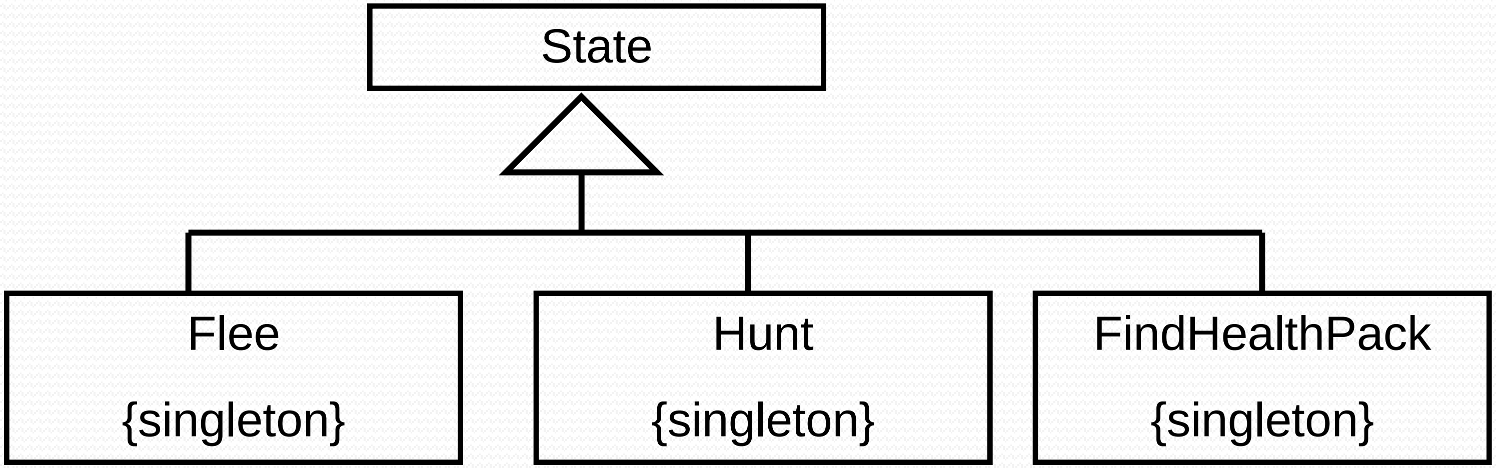
FleeOn()

SeekOff()

FleeOff()

..etc

Reminder


State

Flee

{singleton}

Hunt

{singleton}

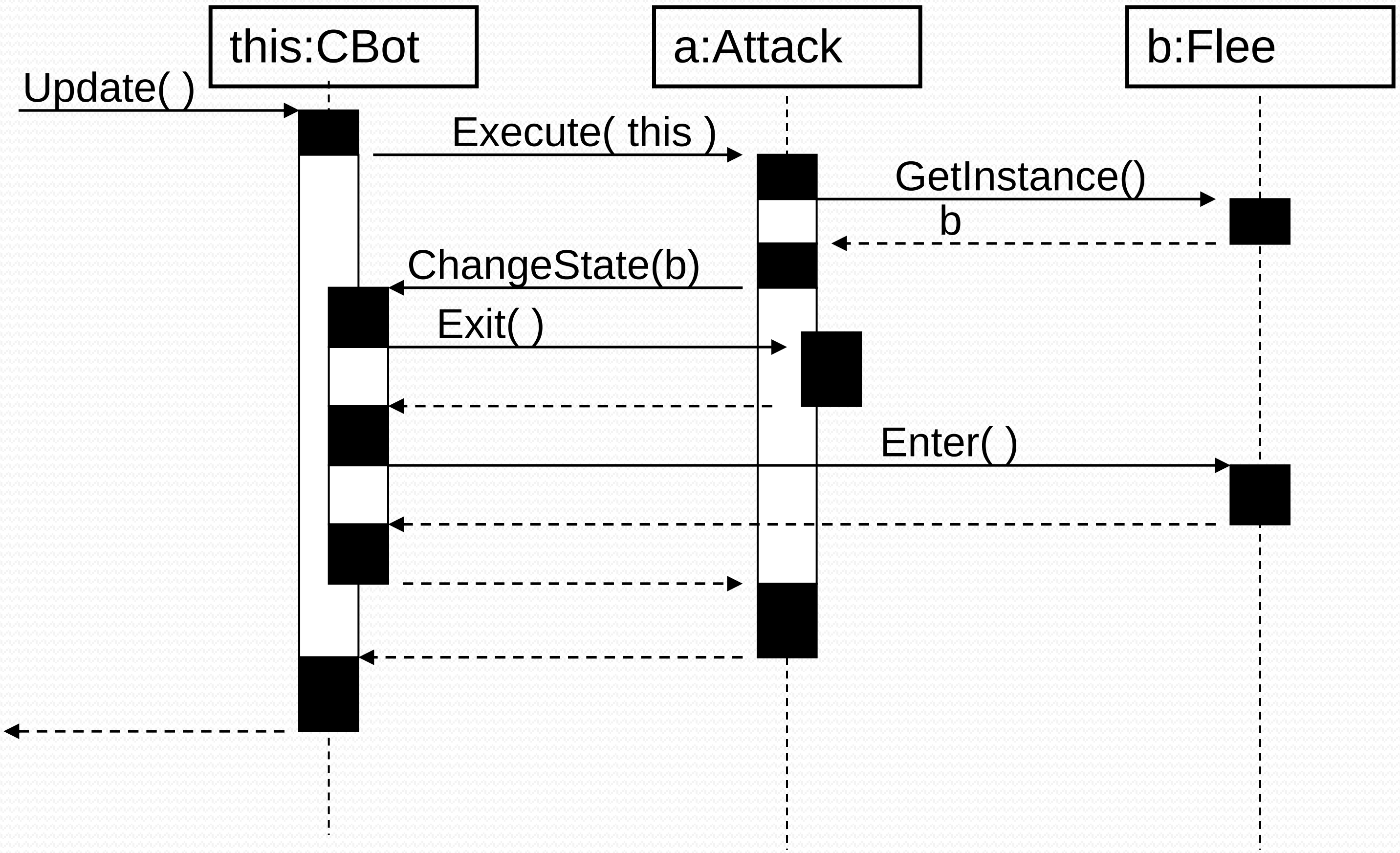
FindHealthPack

{singleton}

Reminder

```
void ChangeState(State newState)
{
  switch(newState)
  {
      case FLEE:
      // Flee entry code
      case HUNT:
      // Hunt entry code
      case GETHEALTHPACK:
      // Followpath entry code
  }
  myState = newState;
}
```

Reminder


this:CBot
a:Attack
b:Flee
Update( )
Execute(this)
GetInstance()
b
ChangeState(b)
Exit( )
Enter( )

Global states

- Sometimes you may have transitions that you want to consider from within a number of states – possibly all of them.
  - Dodge
  - Run away
  - Reload
  - Tumbleweed

Global states

- One solution is to put code to check for these states in the CBot’s Update function.
- Poor cohesion.
- Better is to add the concept of a “global state”.
  - A state that all objects are always in, in addition to their individual state.
  - Call update twice.
- An alternative is to make use of ‘composite states’.
- Trickier to implement, but can use **Boost.Statechart**.

Global states

```
class CBot
{
  State* mpCurrentState;
  State* mpPreviousState;
  State* mpGlobalState;
};
```

Global states

```
void Update()
{
  mpGlobalState->Execute(this);
  mpCurrentState->Execute(this);
}
```

Global states

```
class Global:public State
{
 void Execute(CBot* pBot)
 {
     if(EnemyAiming())
         pBot->ChangeState(Dodge::instance);
 }
};
```

Global states

- Often, these checks are for short emergency actions.
- Once they are done, the bot will return to its previous state.

Global states

```
class Dodge: public State
{
  void Execute(CBot* pBot)
  {
     if( /*safe*/ )
     pBot->ReturnToPreviousState();
  }
};
```

Global states

```
void CBot::ReturnToPreviousState()
{
  if(mpPreviousState)
       ChangeState(mpPreviousState)
  else
       {
             DebugMessage(“Changing to null state”);
             ChangeState(Tumbleweed::instance);
       }
}

// Danger of oscillation?
```

Tactical evaluation

- In many states you will want to write functions like:

```
bool StrongerThanEnemy( );
bool NeedHealing( );
bool EnemyNearMyFlag( );
int BestTarget( );
int MostDangerousEnemy( );
```

Tactical evaluation

- It is tempting to put them in the State superclass.
- What problem will this cause?
- Normally, they would just go into each state.
  - Duplication?
  - Is this good or bad?

Tactical evaluation

- Often these functions use lots of tinkerable values.
- For example, finding the most important threat uses:
    - Range to target.
    - Current state of target.
    - Target's ammunition.
    - Target's range to flag.
    - Target's range to its target.
    - Who is target aiming at?
    - It is my current target?
    - And more.

Tactical evaluation

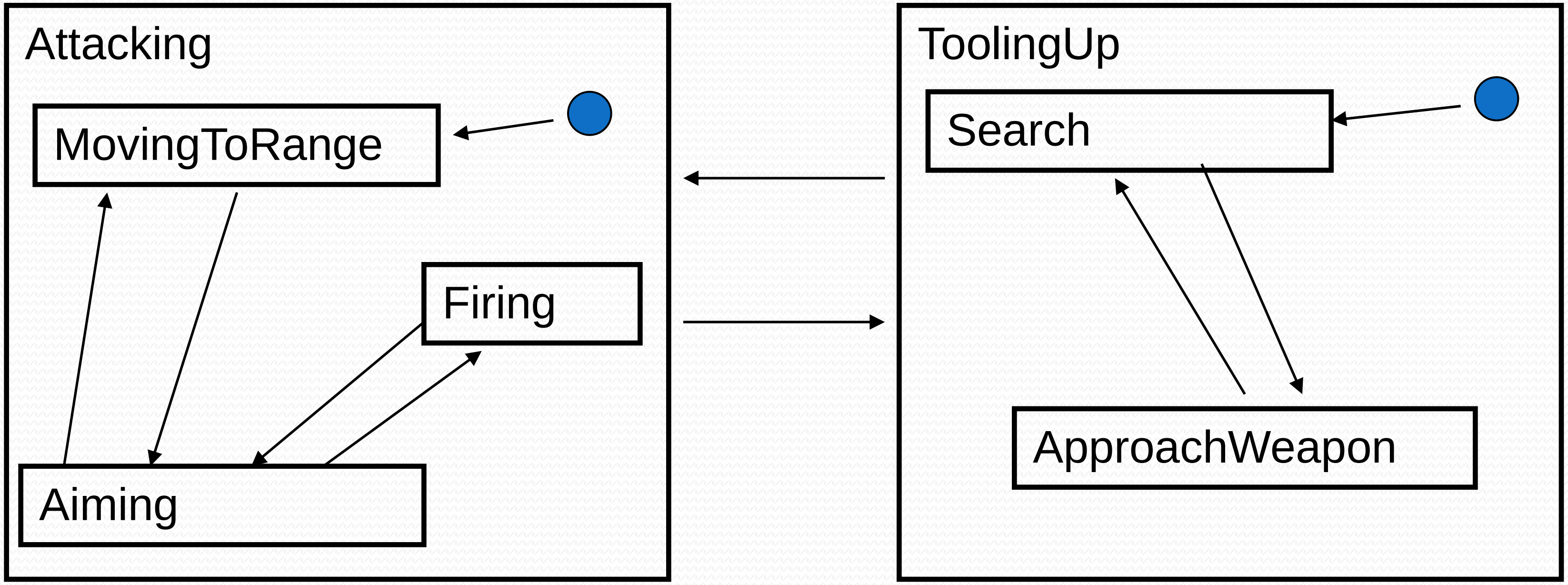
- Your evaluation function will add up a priority for each target, using numbers based on these factors.
- The one with the largest value is the most important target.
  - Note this is not something that you want to do each frame.

Tactical evaluation

- The weighting of each factor is often unknown.
  - Plus they are interdependent.
- Fortunately, they are not too dependent on precision.
- But still take a lot of tuning.
  - Good candidate for scripting (at least initially).

Superstates

- Often you have states within states.



Superstates

- One easy way is to have an extra pointer in the CBot.

```
class CBot
{
  State* mpMajorState;
  State* mpMinorState;
  State* mpPreviousState;
  State* mpGlobalState;
};
```

Superstates

- The major state may have Entry( ) and Exit( ), but rarely needs Execute( ).
  - The minor state will do the individual Execute( ) actions.
- One way to do this is to allow the CBot to call both Entry( ) actions on change state, and the Execute( ) of the minor state each frame.
- Problems?

Superstates

- You won’t always have a minor state.
- A better way is to run Execute for the major state. If the major state knows there will be substates, IT will run the minor state’s Execute.

Superstates

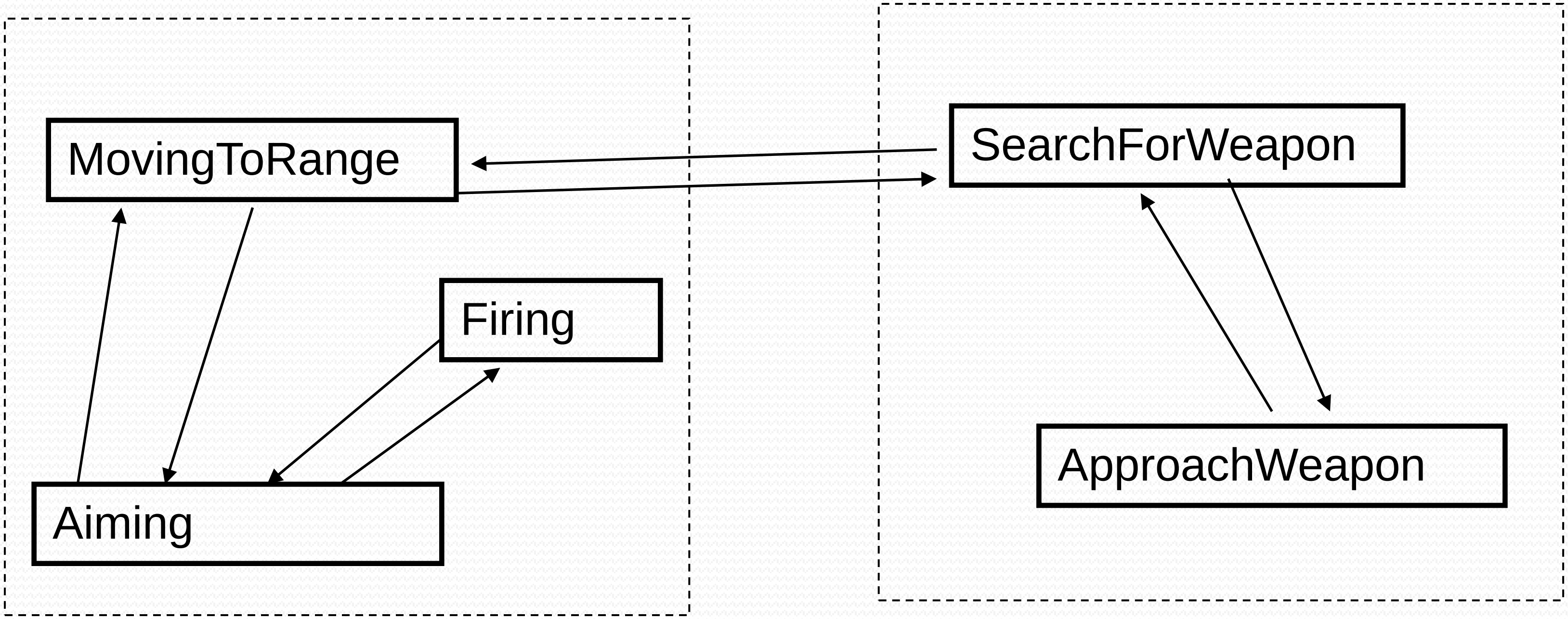
```
void Attack::Execute(CBot* pTheBot)
{
  pTheBot->mpMinorState->Execute();
}

// How does the State get access to the
// private mpMinorState?
```

Superstates

- Still a problem – only limited to two levels of encapsulation.
    - Implement the mpMajorState and mpMinorState as a stack.
    - Pass the stack each time to the state's execute method.
- Now ChangeState becomes very complicated.
    - And whole system is a bit slow.

Superstates

- An alternative, simpler system is to just stick with a single level of states.



Superstates

- It’s still possible to have very different styles and approaches to each bucket of states.
- Can easily be written by different people.
    - Must have a clear entry state.
    - Possibly on that just decides which state to actually be in.
    - Make use of global state to move out of the bucket.

Messages

- Often your Bots will want to work together.
  - One will be using suppressing fire.
  - One will be sniping.
  - One will be making a dash for the flag.
- One way is for the bots to be constantly monitoring each other.
  - For example if you see that the bot going for the flag is killed, you can take over that role.

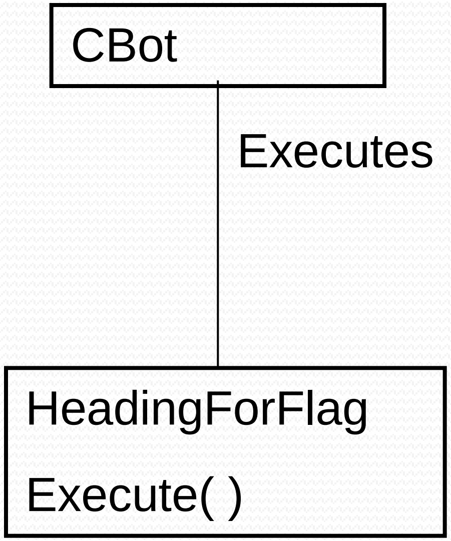
Messages

- It is usually better to use messages.
  - More event-driven.
  - Better encapsulation.
  - Faster.
- The bot that was grabbing the flag sends a message that he has just died.
  - A reciepient of the message will see if he is the closest and take over.

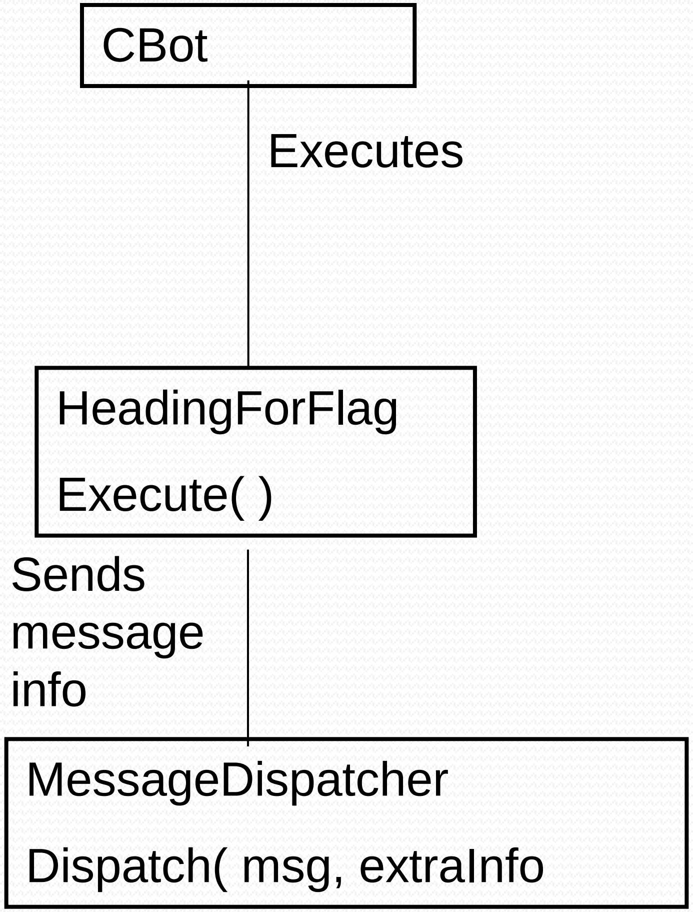
Messages

```
struct Message
{
  int miSenderID;
  int miReceiverID;
  MessageType mMsg;
  double mdWaitUntil;
  void* pExtraInfo;
};
```

Messages



Messages



CBot

Executes

HeadingForFlag

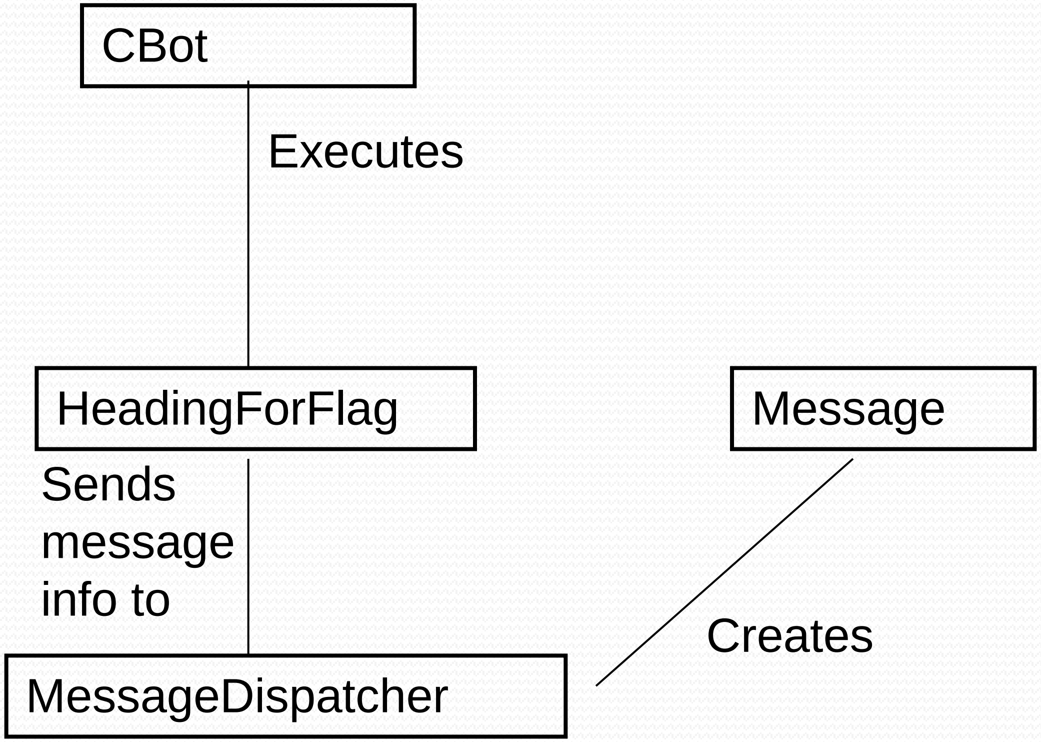
Execute( )

Sends message info

MessageDispatcher

Dispatch(msg, extraInfo

Messages



CBot

Executes

HeadingForFlag

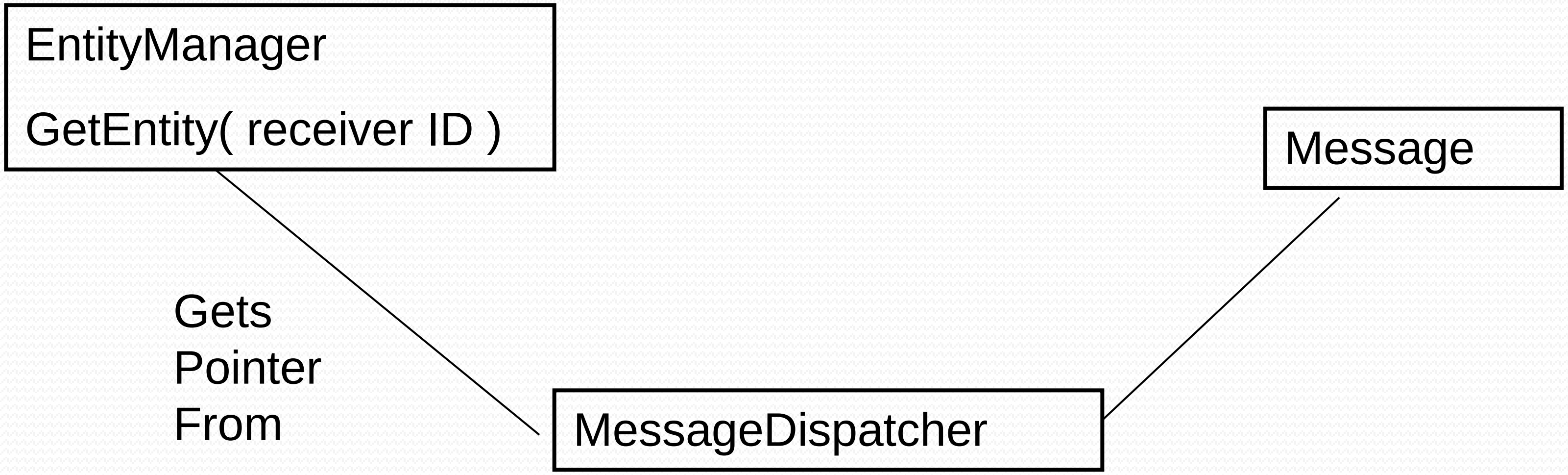
Sends message info to

MessageDispatcher

Creates

Message

Messages



EntityManager

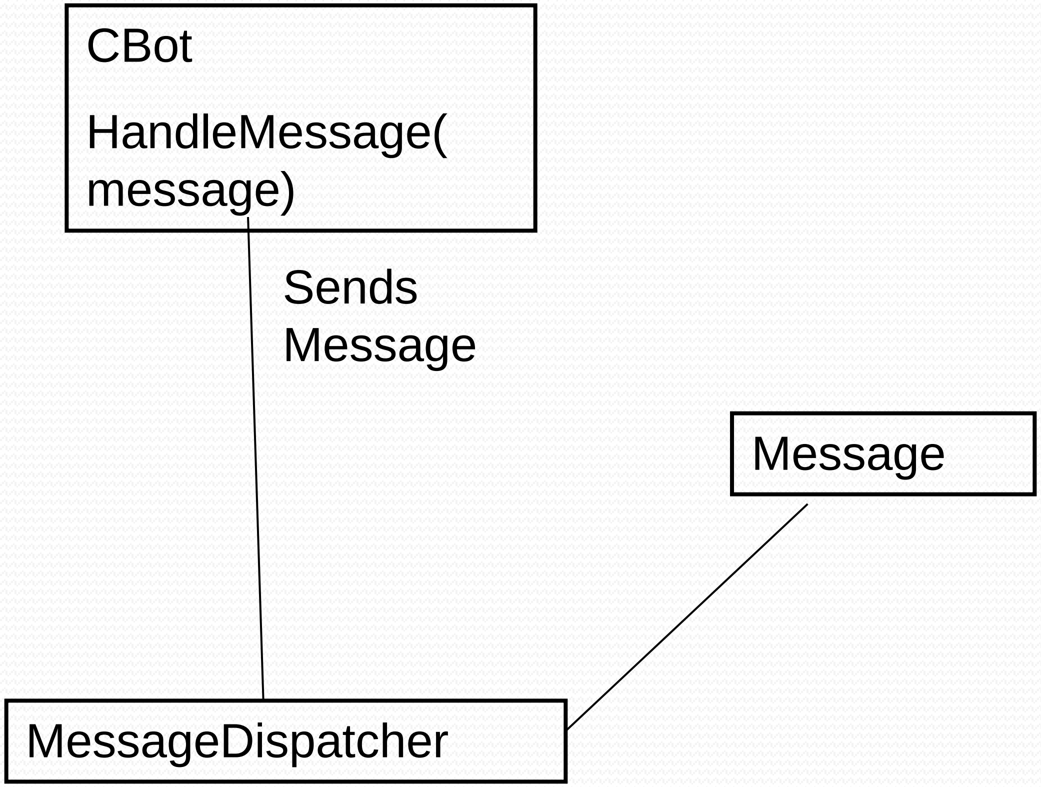
GetEntity(receiver ID)

Gets
Pointer
From

MessageDispatcher

Message

Messages



CBot

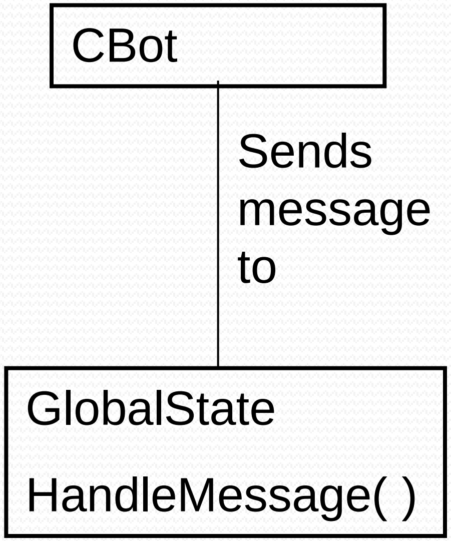
HandleMessage(message)

Sends Message

Message

MessageDispatcher

Messages



CBot

Sends message to

GlobalState

HandleMessage( )

Summary

- Reminder
  - Global states for common stuff
  - Tactical evaluation and tinkering
  - Superstates
    - Hard way
    - Very hard way
    - Simple way
  - Messaging system (basic introduction)

Next week

- Scripting
    - Lua