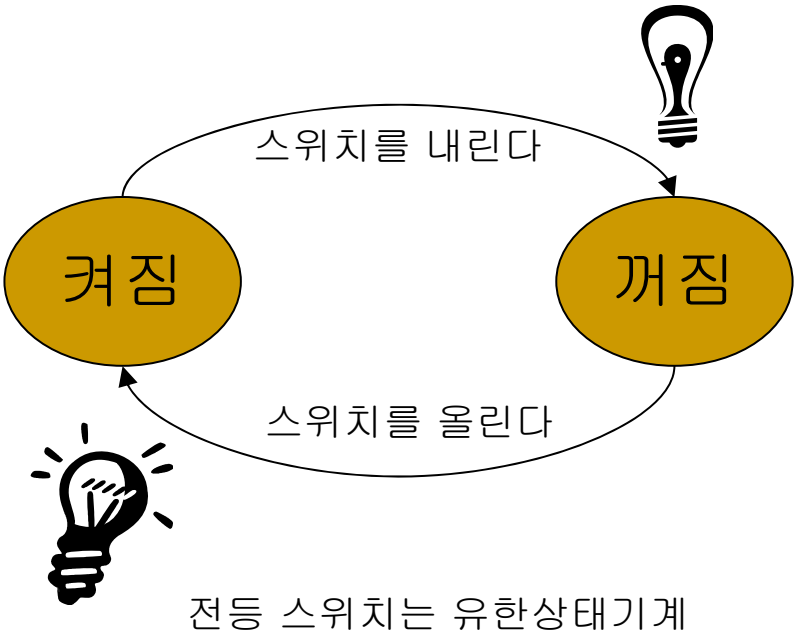
Chapter 2

상태구동형 에이전트의 디자인



전등 스위치는 유한상태기계

유한 상태 기계란 무엇인가?

- 주어지는 모든 시간에서 처할 수 있는 유한 개의 상태를 가지고
- 주어지는 입력에 따라
  - 어떤 상태에서 다른 상태로 전환시키거나
  - 출력이나 액션이 일어나게 하는
- 장치 또는 그런 장치를 나타낸 모델

유한 상태 기계

- Finite State Machine (FSM)의 장점
  - 빠르고 코딩하기가 쉽다
  - 오류수정이 용이하다
  - 계산 부담이 없다
  - 직관적이다
    - 게임 에이전트의 행동을 여러 개의 상태들로 분해
    - 이것들을 조작하는 데 요구되는 규칙들을 세우는 것
  - 유연성이 있다

유한상태기계 구현하기

- 상태를 나타내는 열거형을 사용
  - if-then, switch

```
enum StateType {RunAway, Patrol, Attack};
void Agent::UpdateState(StateType CurrentState) {
    switch(CurrentState) {
      case RunAway:          ...
      case Patrol: ...
      case Attack: ...
    }
}
```

  - 유연성이 없고, 확장이 어려움

상태 전환표 (state transition table)

- 조건들과 이 조건들이 이끌어가는 상태들
- 상태 전환표의 예

현재상태	조 건	상태 전환
도망가기	안전하다	순찰하기
공격하기	적보다 약하다	도망가기
순찰하기	위협받고 적보다 강하다	공격하기
순찰하기	위협받고 적보다 약하다	도망가기

상태 디자인 패턴 구조(내장된 규칙들)

- '상태들 자신의 내부에 상태 전환을 위한 규칙'들을 내장
  - 상태들은 객체로 캡슐화되어 있고
  - 상태 전환을 쉽게 하는데 요구되는 논리들을 포함
    - 예) 상태 전환칩은 역할을 분담해서 카트리지로 규칙들을 직접 이동
  - 모든 상태 객체들은 공통의 interface를 공유
    - State라는 순수 가상 클래스

```
class State
{
    public:
    virtual void Execute(Troll* troll) = 0;
};
```



```
class Troll
{
    State* m_pCurrentState;
public:
    void Update() // 갱신될 때 어떻게 행동할지 현재 상태에
    {     m_pCurrentState->Execute(this); } // 따라 달라짐

    void ChangeState(const State* pNewState)
    {
        delete m_pCurrentState;
        m_pCurrentState = pNewState;
    }
};
```

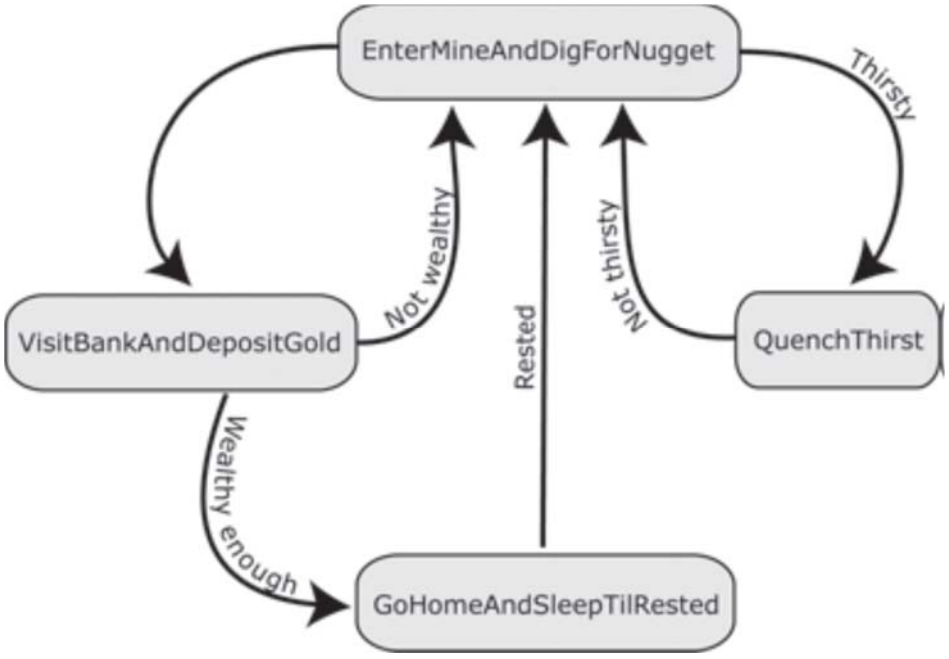
- 상태 구동형(state-driven) 행동을 구현하는 세련된 방법
  - 각 상태에서 액션을 추가하거나 제거하는 작업이 쉬움
  - 각 상태의 Enter와 Exit 메소드를 생성하고 그에 따라 에이전트의 ChangeState 메소드를 조정하는 일뿐

```
//------------State_RunAway
class State_RunAway : public State
{
public:
  void Execute(Troll* troll)
  {
    if (troll->isSafe())    {
      troll->ChangeState(new
     State_Sleep());
    }
    else   {
      troll->MoveAwayFromEnemy();
    }
  }
};
```

```
//------------State_Sleep
class State_Sleep : public State
{
public:
  void Execute(Troll* troll)
  {
    if (troll->isThreatened())    {
      troll->ChangeState(new
     State_RunAway())
    }
    else    {
      troll->Snore();
    }
  }
};
```


West World 프로젝트

- 구성
  - 네 장소: 금광, 은행, 술집, 행복한 집
  - 등장인물: Bob, Elsa
  - 조건변수: 목마름, 피곤함, 금 갯수
- BaseGameEntity class
  - 모든 거주자들이 파생되는 기초 클래스
  - 순수 가상 멤버 함수 Update() 명시
    - 갱신 단계에서 매번 호출됨
- Miner class
  - BaseGameEntity class에서 파생
  - 건강, 피곤함, 위치 등의 속성 멤버
  - State* m_pCurrentState;
  - void ChangeState(State* new_state);
  - void Update();



광부 Bob의 상태 전환 도표

상태 디자인 패턴

- 각각의 게임 에이전트 상태
  - 고유한 클래스로 구현됨
  - 각 에이전트는 현재 상태의 인스턴스를 가리키는 포인터를 가짐
  - 에이전트는 ChangeState 멤버 함수를 구현
    - 상태 전환 가능
    - 상태 전환 결정을 위한 논리는 state 클래스 내에 포함
  - 각 상태의 추가 method
    - Enter(), Exit() : Miner가 상태를 바꿀 때만 호출됨
      - 상태 진입이나 퇴장 시에만 한 번 실행되는 논리를 코딩

State

- 유일 객체 : http://c2.com/cgi/wiki?SingletonPattern
  - 에이전트가 공유하는 각 상태의 인스턴스가 하나만 존재

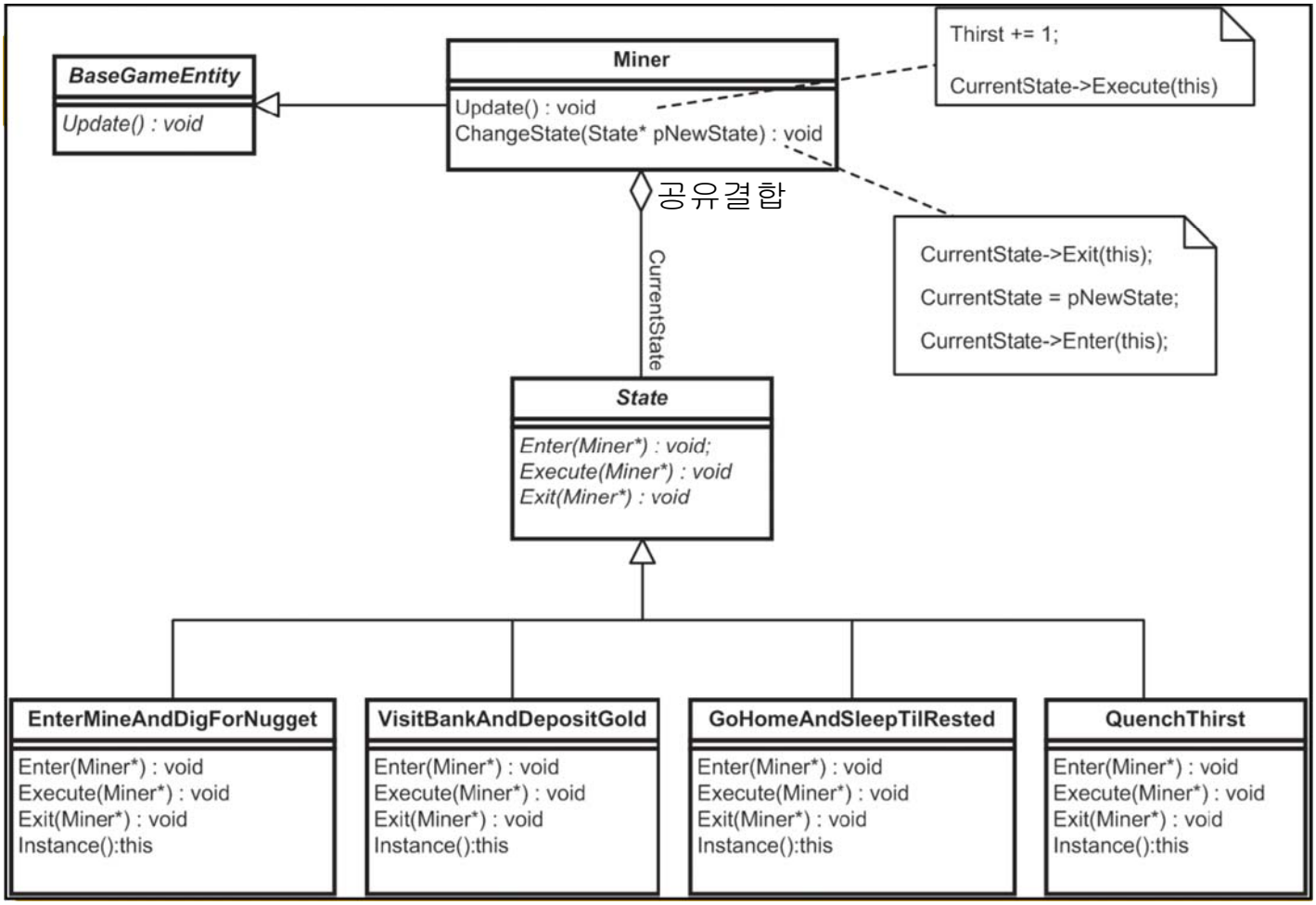
```
class EnterMineAndDigForNugget : public State
{
  //this is a singleton
  static EnterMineAndDigForNugget* Instance();
  virtual void Enter(Miner* miner);
  virtual void Execute(Miner* miner);
  virtual void Exit(Miner* miner);
};
```



```
EnterMineAndDigForNugget*
    EnterMineAndDigForNugget::Instance()
{
  static EnterMineAndDigForNugget instance;
  return &instance;
}
```

클래스 다이어그램

- 결합(association)
  - 클래스 인스턴스간의 연결을 나타냄
  - 실선으로 표시
- 집합(aggregation): 결합의 특수한 경우로 공유(shared)와 혼성(composite)
  - 공유(shared) 결합
    - 빈 다이아몬드로 표시
    - 부분이 전체에 공유될 때
  - 혼성(compotite) 결합
    - 검은 다이이몬드로 표시
    - 부분이 전체에 의해 소유될 때



광부 Bob의 상태기계 구현을 위한 UML 클래스 도표

실습

- Miner에 상태 추가

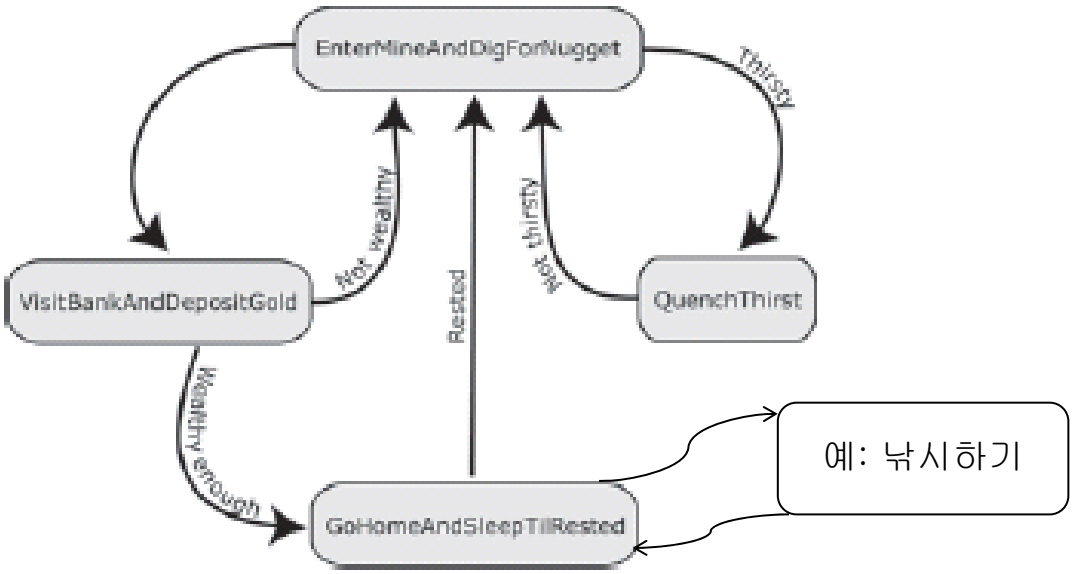


Figure 2.2. Miner Bob's state transition diagram

상태 기본 클래스 재사용

```
class Miner;
class State {
public:
  virtual ~State(){}
  virtual void Enter(Miner*)=0;
  virtual void Execute(Miner*)=0;
  virtual void Exit(Miner*)=0;};
```

- 각 캐릭터가 별개의 기본 State 클래스를 생성해야 함

```
class EnterMineAndDigForNugget :
    public State
{ … };
```

```
template <class entity_type>
class State {
public:
  virtual ~State(){}
  virtual void Enter(entity_type*)=0;
  virtual void Execute(entity_type*)=0;
  virtual void Exit(entity_type*)=0;};
```

• 모든 캐릭터가 기본 State 클래스를 재사용

```
class EnterMineAndDigForNugget :
public State<Miner>
{ … };
```

전역 상태 및 상태 블립

- 전역 상태
  - 어느 때라도 발생할 수 있는 상태
    - Miner::Update에 넣거나 모든 상태에 복사된 조건 논리를 추가하는 것은 안좋은 방법
  - FSM이 갱신될 때마다 호출되는 전역 상태를 생성

```
State<Miner>* m_pGlobalState;
```

- 상태 블립(state blip)
  - 어떤 상태에서 나갈 때, 이전 상태로 복귀한다는 조건하에 에이전트가 어떤 상태에 들어가는 경우의 행동

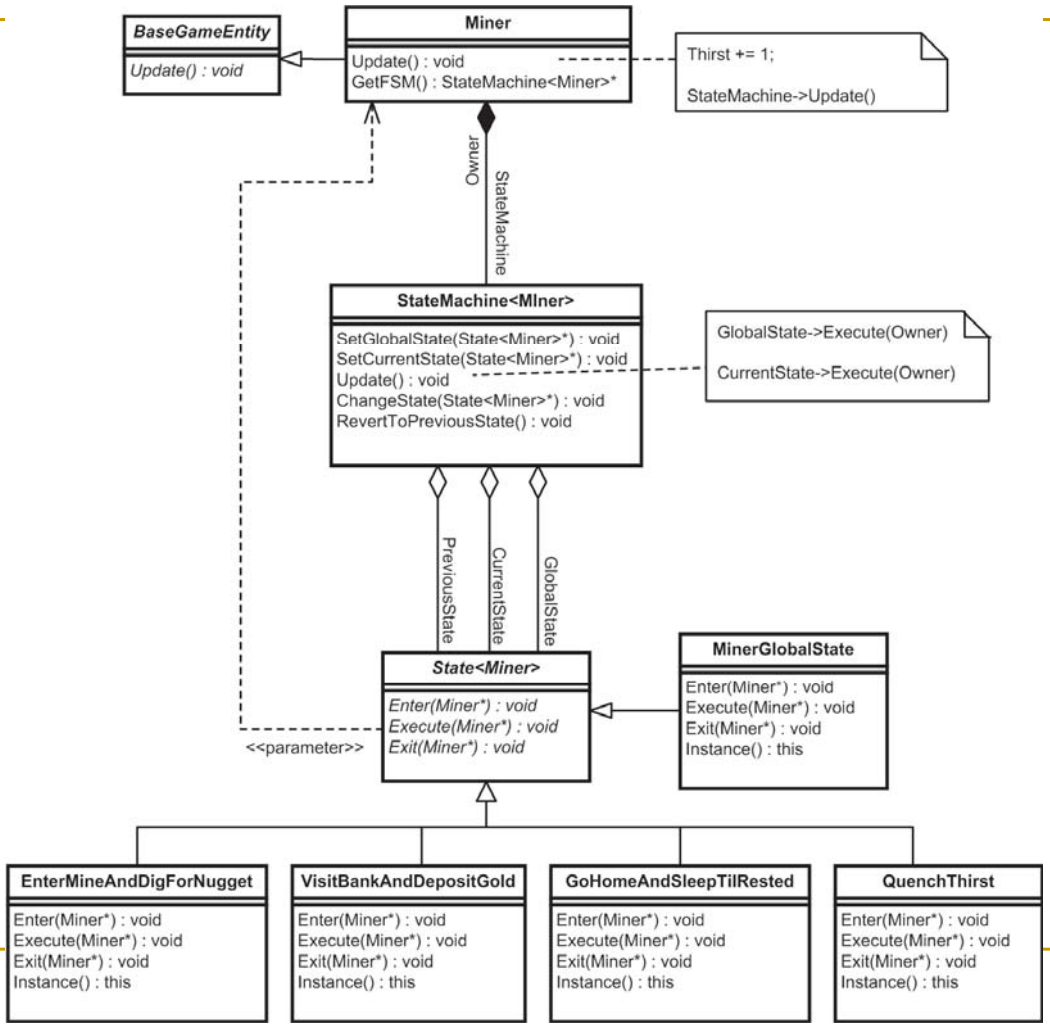
```
State<Miner>* m_pPreviousState;
void RevertToPreviousState( )
    {ChangeState(m_pPreviousState);}
```


상태기계 클래스 생성하기

- 상태와 연관된 모든 데이터와 메소드들을 하나의 상태기계 클래스로 캡슐화

```
template <class entity_type>
class StateMachine
{
    ...
}
```

- 에이전트는 StateMachine의 인스턴스를 소유하고 현재 상태, 전역 변수 및 이전상태의 관리를 위임



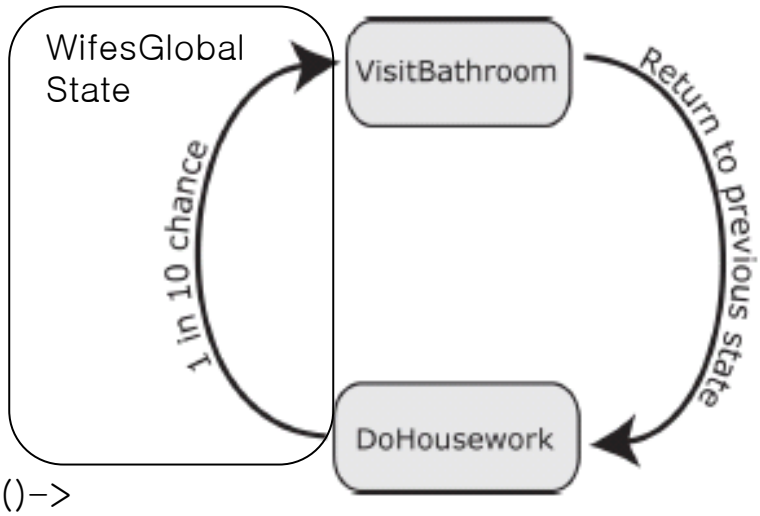
개선된 **Miner** 클래스

```
class Miner : public BaseGameEntity {
  StateMachine<Miner>*  m_pStateMachine;
  Miner(int id):BaseGameEntity(id), m_Location(shack),... m_iFatigue(0)
  {
    m_pStateMachine = new StateMachine<Miner>(this);
    m_pStateMachine->SetCurrentState(GoHomeAndSleepTilRested::Instance());
  }
  StateMachine<Miner>*  GetFSM() const {return m_pStateMachine;}
  void Miner::Update()
  {
    SetTextColor(FOREGROUND_RED| FOREGROUND_INTENSITY);
    m_iThirst += 1;
    m_pStateMachine->Update();
  }
};
```

WestWorldWithWoman

- Elsa

m_pCurrentState == VisitBathroom
m_pPreviousState == DoHouseWork



wife->GetFSM()->
ChangeState(VisitBathroom::Instance());

m_pCurrentState == DoHousework

실습

- 등장인물 추가
  - Bob, Elsa, ?
- 추가 등장인물의 상태들 추가
  - 보통 State와 GlobalState 모두 추가

FSM에 메시지 처리 기능 추가하기

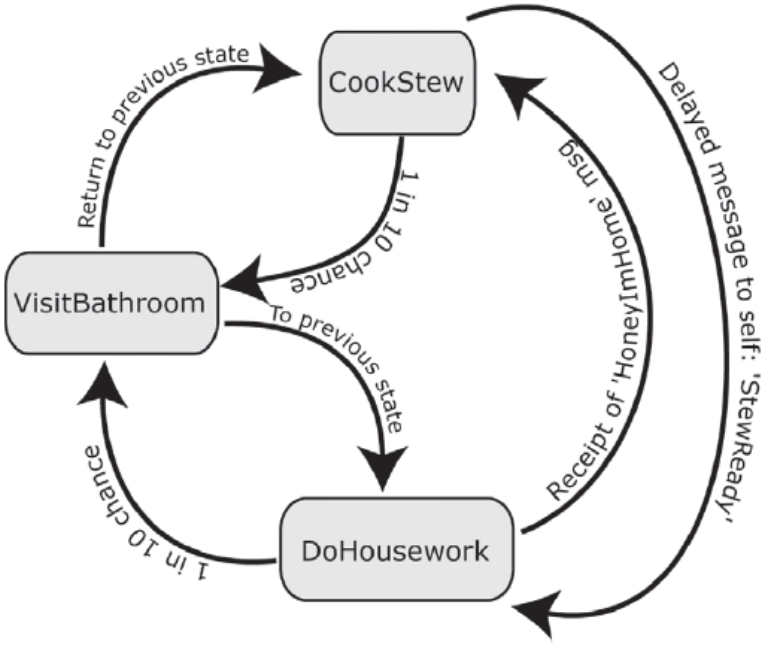
- 이벤트 다루기(event handling)
  - 이벤트 메시지가 자신에게 뿌려질 때까지 단지 자기 업무만 하고 있으면 됨
  - 없으면, 게임세계에서 특별한 액션이 발생했는지를 알기 위해 끊임없이 조사해야 함

전보 구조

- 메시지
  - enumerated type: Msg_HiHoneyImHome, Msg_StewReady
  - Telegram(전보) 구조

```
struct Telegram {
        int         Sender;
        int         Receiver;
        int         Msg;
        double      DispatchTime;
        void*       ExtraInfo;
};
```

광부 Bob과 Elsa의 통신



메시지 급송 및 관리

```
class EntityManager {
private:
  typedef std::map<int, BaseGameEntity*> EntityMap;
  EntityMap m_EntityMap;
public:
  static EntityManager* Instance();
  void            RegisterEntity(BaseGameEntity* NewEntity);
  BaseGameEntity* GetEntityFromID(int id)const;
  void            RemoveEntity(BaseGameEntity* pEntity);
};
```

- MessageDispatcher가 참조하도록 제공되는 인스턴스화된 엔티티들로 구성된 일종의 데이터베이스

MessageDispatcher class

```
class MessageDispatcher {
private:
  std::set<Telegram> PriorityQ;
  void Discharge(BaseGameEntity* pReceiver, const Telegram& msg);
public:
  static MessageDispatcher* Instance();
  //send a message to another agent. Receiving agent is referenced by ID.
  void DispatchMessage(double  delay,
                       int    sender,
                       int    receiver,
                       int    msg,
                       void*  ExtraInfo);
  //send out any delayed messages. This method is called each time through
  //the main game loop.
  void DispatchDelayedMessages();
};
```



```
void MessageDispatcher::DispatchMessage(double  delay,
                                        int    sender,
                                        int    receiver,
                                        int    msg,
                                        void*  ExtraInfo)
{
  //get pointers to the sender and receiver
  BaseGameEntity* pSender   = EntityMgr->GetEntityFromID(sender);
  BaseGameEntity* pReceiver = EntityMgr->GetEntityFromID(receiver);
  //create the telegram
  Telegram telegram(0, sender, receiver, msg, ExtraInfo);
  //if there is no delay, route telegram immediately
  if (delay <= 0.0f)
  {
    cout << "₩nInstant telegram dispatched at time: " << Clock->GetCurrentTime()
         << " by " << GetNameOfEntity(pSender->ID()) << " for "
<< GetNameOfEntity(pReceiver->ID())
         << ". Msg is "<< MsgToStr(msg);
    //send the telegram to the recipient
    Discharge(pReceiver, telegram);
  }
```

```
 //else calculate the time when the telegram should be dispatched
  else
  {
    double CurrentTime = Clock->GetCurrentTime();
    telegram.DispatchTime = CurrentTime + delay;
    //and put it in the queue
    PriorityQ.insert(telegram);
    cout << "₩nDelayed telegram from " << GetNameOfEntity(pSender->ID())
<< " recorded at time "
          << Clock->GetCurrentTime() << " for " << GetNameOfEntity(pReceiver-
>ID())
          << ". Msg is "<< MsgToStr(msg);
  }
}
```



```
void MessageDispatcher::DispatchDelayedMessages()
{
  double CurrentTime = Clock->GetCurrentTime();
  while( !PriorityQ.empty() &&
        (PriorityQ.begin()->DispatchTime < CurrentTime) &&
        (PriorityQ.begin()->DispatchTime > 0) )
  {
    const Telegram& telegram = *PriorityQ.begin();
    BaseGameEntity* pReceiver = EntityMgr->GetEntityFromID(telegram.Receiver);
    cout << "₩nQueued telegram ready for dispatch: Sent to "
        << GetNameOfEntity(pReceiver->ID()) << ". Msg is "
<< MsgToStr(telegram.Msg);
    Discharge(pReceiver, telegram);
    PriorityQ.erase(PriorityQ.begin());
  }
}
```

```
void MessageDispatcher::Discharge(BaseGameEntity* pReceiver,
                                  const Telegram& telegram)
{
  if (!pReceiver->HandleMessage(telegram))
  {
    //telegram could not be handled
    cout << "Message not handled";
  }
}
```

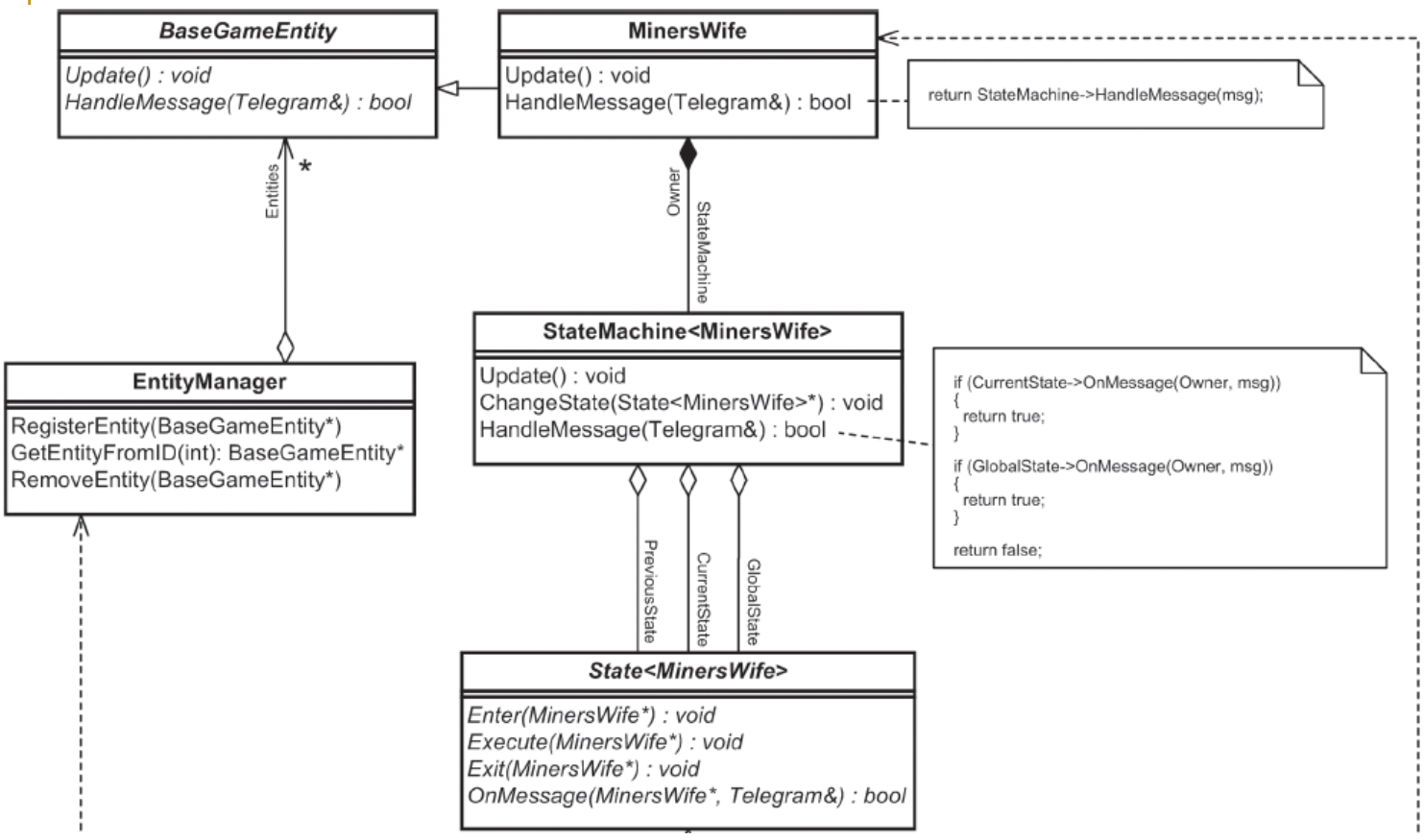

메시지 처리하기

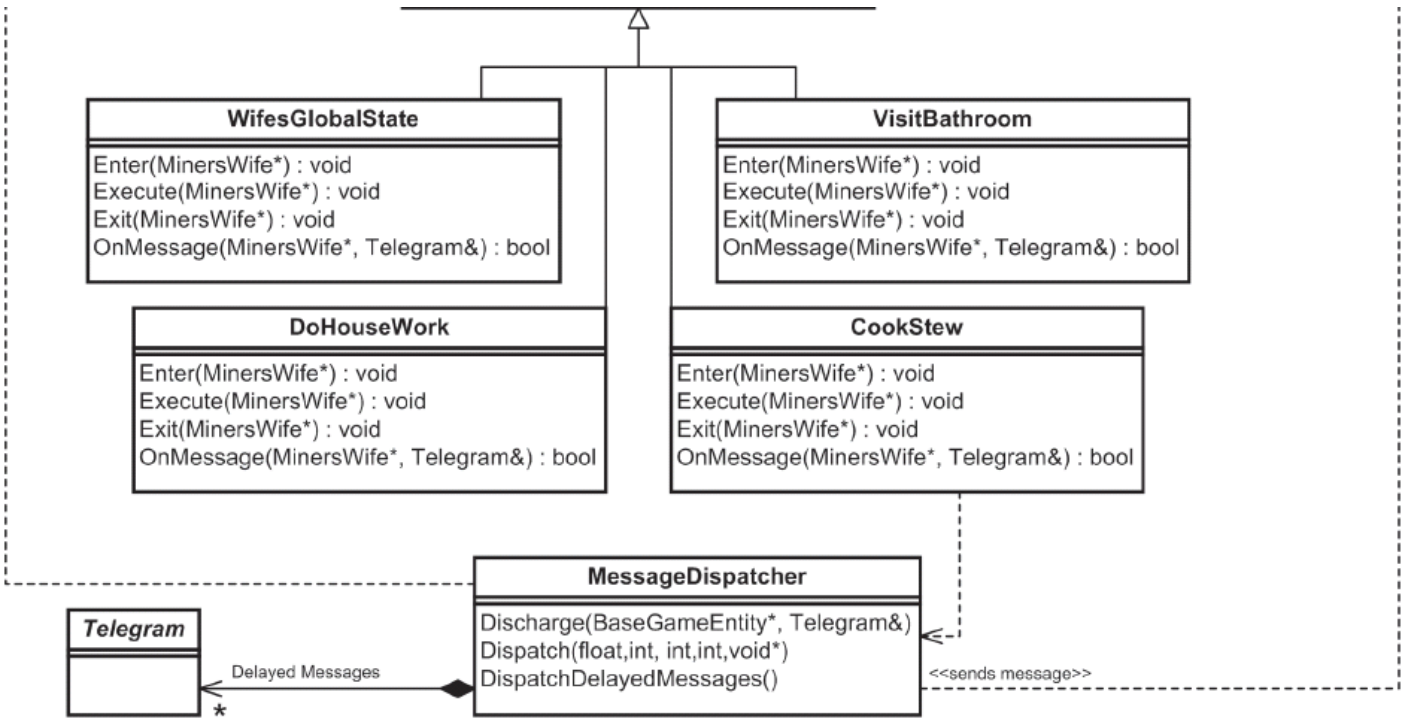
```
class BaseGameEntity
{
  int          m_ID;
  virtual void  Update()=0;
  //all entities can communicate using messages. They are sent
  //using the MessageDispatcher singleton class
  virtual bool  HandleMessage(const Telegram& msg)=0;
  int          ID()const{return m_ID;}
};
 class State
 {
 public:
   virtual void Enter(entity_type*)=0;
   virtual void Execute(entity_type*)=0;
   virtual void Exit(entity_type*)=0;
   //this executes if the agent receives a message from the
   //message dispatcher
   virtual bool OnMessage(entity_type*, const Telegram&)=0;
 };
```

```
bool  StateMachine::HandleMessage(const Telegram& msg)const
 {
   //first see if the current state is valid and that it can handle
   //the message
   if (m_pCurrentState && m_pCurrentState->OnMessage(m_pOwner, msg))
   {
     return true;
   }

   //if not, and if a global state has been implemented, send
   //the message to the global state
   if (m_pGlobalState && m_pGlobalState->OnMessage(m_pOwner, msg))
   {
     return true;
   }

   return false;
 }
```



실습

- WestWorldWithMessaging
  - Bob과 Elsa 통신 분석
    - 메시지가 없어질 수 있는 상황 파악
  - 메시지 추가
    - 보통 상태에서 처리되는 메시지
    - GlobalState에서 처리되는 메시지

과제

- FSM With Messaging
  - 새로운 시나리오 구현: 새로운 등장인물 3명
  - Global 상태, 메시지 처리 포함
    - 등장인물 모두
    - 보통 State와 GlobalState , 메시지 처리 모두 구현
  - 제출물
    - 상태도표 다이어그램(pptx)
    - 실행 캡처(학번, 성명 포함) 화면
    - 소스코드(학번, 성명 포함, debug 폴더와 .vs 폴더 삭제)