**항목 35. 가상 함수 대신 쓸 것들도 생각해 두는 자세를 시시때때로 길러 두자**

캐릭터의 체력이 얼마나 남았는지를 나타내는 정수 값을 반환 하는 healthValue라는 이름의 멤버 함수를 제공 하기로 했으며 이 함수를 가상 함수로 선언하는 것이 확실한 설계가 될 것이다.

|  |
| --- |
| class GameCharater {  public :  virtual int healthValue() const; // 캐릭터의 체력치를 반환하는 함수로서,  ... // 파생 클래스는 이 함수를 재정의 할 수 있다.    }; |

healthValue 함수가 순수 가상 함수로 선언되지 않은 것으로 보아, 체력치를 계산하는 **기본 알고리즘이 제공**된다는 사실을 알 수 있다.(항목 34) 당연한 설계 이지만, 어떤 의미로는 약점이 된다. 이것 말고 적당한 다른 방법을 떠올리는 것도 잘 안 된다.

**1. 비가상 인터페이스 관용구를 통한 템플릿 메서드 패턴**

“**가상 함수는 반드시 private 멤버로 두어야 한다**”고 주장하는 소위 **“가상 함수 은폐론**”으로 시작하자. 이 이론을 따르면 더 괜찮은 설계는 healthValue를 public 멤버 함수로 그대로 두되 비 가상 함수로 선언하도, 내부적으로는 실제 동작을 맡은 private 가상 함수를 호출하는 식으로 만드는 것.

|  |
| --- |
| class GameCharater {  public:  int healthValue() const // 파생 클래스는 이제 이 함수를 재정의할 수 없습니다.  {  ... // "사전" 동작을 수행합니다. 아래를 봐 주세요.  int retVal = doHealthValue(); // 실제 동작을 수행합니다.  ... // "사후" 동작을 수행합니다. 아래를 봐 주세요  return retVal;  }  ...  private:  virtual int doHealthValue() const // 파생 클래스는 이 함수를 재정의할 수 있습니다.  {  ... // 캐릭터의 체력치 계산을 위한 기본 알고리즘 구현  }  }; |

멤버 함수의 본문이 클래스 정의 안에 들어가 있다. 이렇게 하면 암시적으로 인라인 함수로 선언된다.(항목 30)

사용자로 하여금 public 비가상 멤버 함수를 통해 private 가상 함수를 간접적으로 호출하게 만드는 방법으로, 비가상 함수 인터페이스(Non-Virtual Interface : NVI) 관용구라고 많이 알려져 있다. **가상 함수의 래퍼(Wrapper)**

NVI 관용구의 이점은 위의 코드에 주석 문으로 써져 있는 “사전 동작”, “사후 동작”에 있다. 주석 문이 가리키는 부분은 실제 동작을 수행하는 가상 함수를 호출하기 전과 호출한 후에 호출될 코드들이 들어갈 자리이다. 즉, 가상 함수가 호출되기 전에 어떤 상태를 구성하고 가상 함수가 호출된 후에 그 상태를 없애는 작업이 랩퍼를 통해 공간적으로 보장된다는 뜻이다.

**2. 함수 포인터로 구현된 전략 패턴**

NVI 관용구는 public 가상 함수를 대신할 수 있는 꽤 괜찮은 방법이지만, 클래스 설계의 관점에서 보면 눈 속임이나 다름 없다. 어쨌든 체력치를 계산하는데 가상 함수를 쓰는 데는 여전하기 때문이다. 조금 더 극적인 설계로 가면 체력의 계산이 구태여 어떤 캐릭터의 일부일 필요가 없다는 것이다. 예로, 캐릭터의 생성자에 체력치 계산용 함수포인터를 넘길 수 있을 것이다.

|  |
| --- |
| class GameCharater; // 전방 선언  // 체력치 계산에 대한 기본 알고리즘을 구현한 함수  int defaultHealthCalc(const GameCharater& gc);  class GameCharacter {  public:  typedef int (\*HealthCalcFunc)(const GameCharacter&);  explicit GameCharacter(HealthCalcFunc hcf = defaultHeathCalc) : healthFunc(hcf) {}  int healthValue() const  { return healthFunc(\*this); }  ...  private:  HealthCalcFunc healthFunc;  }; |

디자인 패턴의 전략(Strategy) 패턴을 응용한 예이다. 이처럼 GameChatater 클래스 계통에 가상 함수를 심는 방법은 재밌는 융통성을 가진다.

**- 같은 캐릭터 타입으로부터 만들어진 객체들도 체력치 계산 함수를 각각 다르게 가질 수 있다.**

|  |
| --- |
| class EvilBadGuy : public GameCharacter {  public:  explicit GameCharacter(HealthCalcFunc hcf = defaultHeathCalc) : healthFunc(hcf) {...}  ...  };  // 다른 동작 원리로 구현된 체력치 계산 함수들  int loseHealthQuickly(const GameCharacter&);  int loseHealthSlowly(const GameCharacter&);    // 같은 타입인테도 체력치 변화가 다르게 나오는 캐릭터들  EvilBadGuy ebg1(loseHealthQuickly);  EvilBadGuy ebg2(loseHealthSlowly); |

**- 게임이 실행되는 런타임 중에 특정 캐릭터에 대한 체력치 계산 함수를 변경할 수 도 있다.**

예를 들어, setHealthCalculator 라는 멤버 함수를 제공하고 체력치 계산 함수를 교체할 수 있다.

이 방법의 단점으로, 체력치 계산 함수가 이제 GameCharater 클래스 계통의 멤버 함수가 아니라는 것, 체력치가 계산되는 개상 객체의 비공개 데이터는 이 함수로 접근할 수 없다는 뜻이다. 캐릭터의 public 인터페이스로 얻은 정보만을 사용해서 캐릭터의 체력치를 계산할 수 있게 되어 있다면 문제가 없겠지만, 정확한 체력치 계산을 위해서 public 멤버가 아닌 정보를 써야할 경우 문제가 생긴다.

Public 영역에 없는 부분을 비멤버 함수로 접근할 수 있게 하려면 그 클래스의 캡슐화를 약화 시켜야 한다는 것이 일반적인 방법이다. 프렌트 선언, 세부 구현 사항의 접근자를 public으로 제공하는 일 등

함수 포인터를 통해 얻은 이점들(객체별로 체력치 계산 함수를 둘 수 있다는 점, 런타임중 교체 가능)이 과연 GameCharacter 클래스의 캡슐화를 떨어뜨리면서 얻은 불이익을 채워 줄지는 사용자의 판단의 몫이다.

**3. th1::function으로 구현한 전략 패턴**

함수 포인터 기반의 방법이 뭔가 꽉 막혀 보일 수 있다. “체력치 계산을 왜 꼭 함수가 해야 해?” 그냥 함수 처럼 동작하는 다른 놈을 쓰면 안되나?라고 반박하고 싶다. Tr1::function 타입의 객체를 써서 기존의 함수 포인터를 대신하게 만드는 순간 이 모든 것이 해결된다. Tr1::function 계열의 객체는 함수호출성 개체(callable entity)(풀어서 말하면 함수포인터, 함수 객체 혹은 멤버 함수 포인터)를 가질 수 있고, 이들 개체는 주어진 시점에서 예상되는 시그니처와 호환되는 시그너처를 갖고 있다.

|  |
| --- |
| class GameCharacter; // 이전과 같습니다.  int defaultHealthCalc(const GameCharater& gc); // 이전과 같습니다.  class GameCharacter {  public:  // HealthCalcFunc는 함수 호출성 개체로서, GameCharacter와 호환되는  // 어떤 것이든 넘겨받아서 호출될 수 있으며 int와 호환되는 모든 타입의  // 객체를 반환합니다. 자세한 내용은 아래를 보시면 나옵니다.  **typedef std::tr1::function<int (const GameCharater&)> HealthCalcFunc;**  explicit GameCharacter(HealthCalcFunc hcf = defaultHeathCalc) : healthFunc(hcf) {}  int healthValue() const  { return healthFunc(\*this); }  ...  private:  HealthCalcFunc healthFunc;  }; |

HealthCalcFunc는 tr1::Function 템플릿을 인스턴스화한 것에 대한 typedef 타입이다. 다시 말해 이 타입은 일반화된 함수 포인터 타입처럼 동작한다는 것이다.

Tr1::function을 인스턴스화하기 위해 매개변수로 쓰인 “대상 시그너처”가 보일 것이다. 이 목적 시그너처를 그대로 읽으면 “const GameCharacter”에 대한 참조자를 받고 int를 반환하는 함수이다. 이렇게 정의된 tr1::function타입으로 만들어진 객체는 앞으로 목적 시그너처와 호환되는 함수호출성 개체를 어떤 것도 가질 수 있다. 앞 절의 설계(GameCharater가 함수 포인터를 물게 했던)와 비교하면 지금 설꼐도 크게 다른 것이 없다. 다른 점이 있다면 GameCharater가 이제는 tr1::function의 객체, 즉 좀더 일반화된 함수 포인터를 물게 되었다는 것.

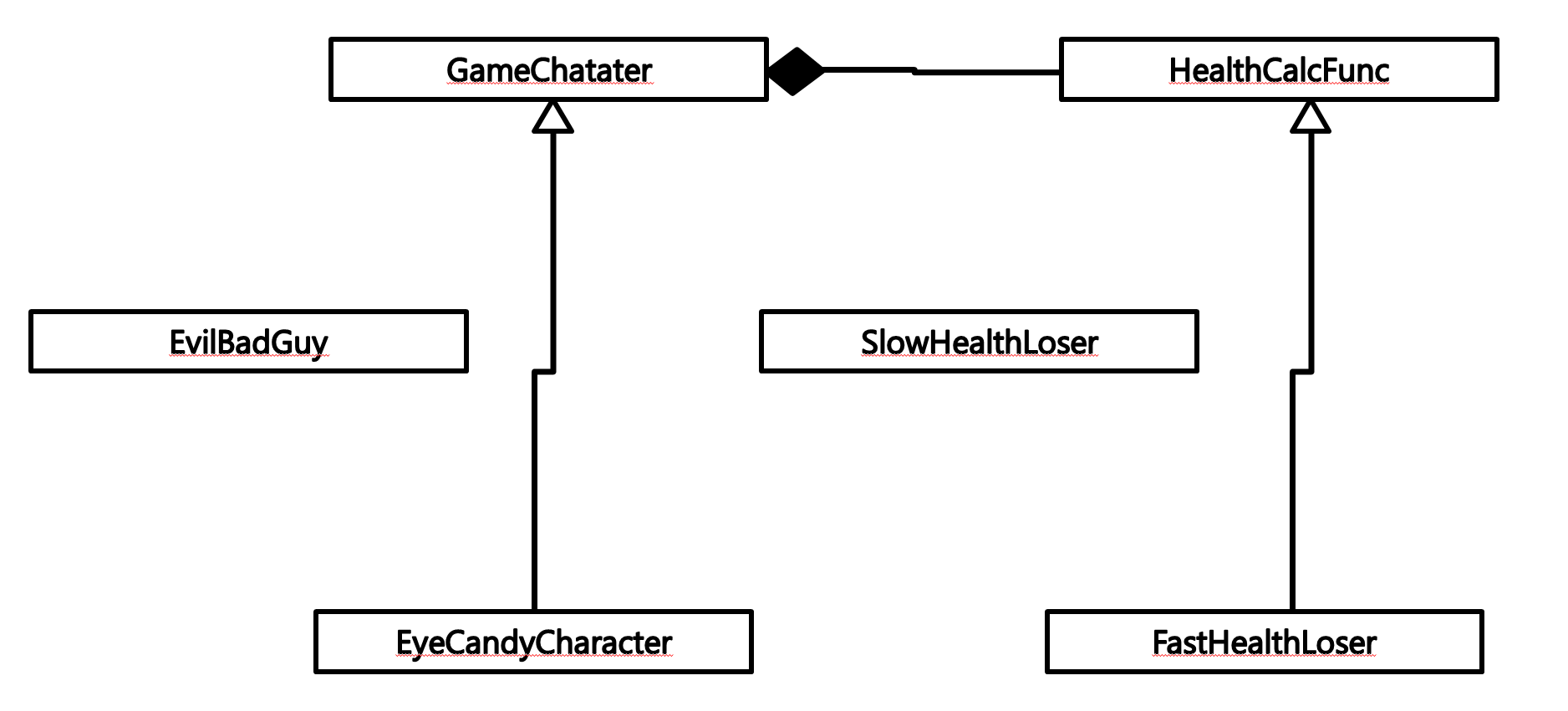
|  |
| --- |
| class EvilBadGuy : public GameCharacter {  public:  explicit GameCharacter(HealthCalcFunc hcf = defaultHeathCalc) : healthFunc(hcf) {...}  ...  };  // 다른 동작 원리로 구현된 체력치 계산 함수들  int loseHealthQuickly(const GameCharacter&);  int loseHealthSlowly(const GameCharacter&);    // 같은 타입인테도 체력치 변화가 다르게 나오는 캐릭터들  EvilBadGuy ebg1(loseHealthQuickly);  EvilBadGuy ebg2(loseHealthSlowly);    tr1::function으로 구현한 전략 패턴  class GameCharacter; // 이전과 같습니다.  int defaultHealthCalc(const GameCharater& gc); // 이전과 같습니다.  class GameCharacter {  public:  // HealthCalcFunc는 함수 호출성 개체로서, GameCharacter와 호환되는  // 어떤 것이든 넘겨받아서 호출될 수 있으며 int와 호환되는 모든 타입의  // 객체를 반환합니다. 자세한 내용은 아래를 보시면 나옵니다.  typedef std::tr1::function<int (const GameCharater&)> HealthCalcFunc;  explicit GameCharacter(HealthCalcFunc hcf = defaultHeathCalc) : healthFunc(hcf) {}  int healthValue() const  { return healthFunc(\*this); }  ...  private:  HealthCalcFunc healthFunc;  };    short calcHealth(const GameCharacter&); // 체력치 계산 함수입니다.  // 반환 타입이 int가 아닌 부분에 주목!  struct HealthCalculator { // 체력치 계산용  int operator() (const GameCharater&) const { ... } // 함수 객체 만들기  }; // 위한 쿨래스    class GameLevel {  public: // 체력치 계산에 쓰일 멤버  float health(const GameCharacter&) const; // 함수입니다. 반환 타입이  ... // int가 아닌 부분에 주목하세요.  };  class EvilBadGuy : public GameCharater { // 예전의 그것과 같습니다.  ...  };  class EyeCandyCharacter : public GameCharater { // 또 하나의 캐릭터 타입.  ... // 생성자는 EvilBadGuy의  }; // 그것과 똑같다고 가정합니다.  EvilBadGuy ebg1(calcHealth); // 체력치 계산을 위한  // 함수를 사용하는 캐릭터  EyeCandyCharacter ecc1(HealthCalculator()); // 체력치 계산을 위한  // 함수 객체를 사용하는 캐릭터  GameLevel currentLevel;  ...  EvilBadGuy egb2( // 체력치 계산을 위한 멤버 함수를  std::tr1::bind(&GameLevel::health, currerntLevel, \_1) // 사용하는 캐릭터 입니다. |

이런 코드를 만들 수 있는 것은 tr1::function 템플릿 덕분이다. 위의 정의문이 말하는 바는 ebg2의 체력치를 계산학 위해 GameLevel 클래스의 health 멤버 함수를 써야 한다는 것이다. 현재, GameLevel::health 함수는 매개변수 하나를 받는 것으로 선언되어 있지만, 실제로는 두 개를 받는다. GameLevel 객체 하나를 암시적으로 받아들이니까 말이다. 이 객체는 this 포인터가 가리키는 것이다.

Ebg2의 체력치 계산에 GameLevel::health 함수를 쓰려고 한다면, 어떻게든 “때려 맞추어야 할 것이다” 매개 변수 두 개를 받는 함수를 매개변수 한 개만 받는 함수로 바꿔야 한다는 것이다. 지금의 예제 코드에서는 ebg2의 체력치 계산에 쓸 GameLevel 객체로서 currentLevel만을 쓸 생각이므로, 우리는 GameLevel::health 함수가 호출될 때마다 currentLevel이 사용되도록 묶어 준것이다. Tr1::bind는 바로 이 묶기 작업을 맡았다. 즉, ebg2의 체력치 계산 함수는 항상 currentLevel만을 GameLevel 객체로 쓴다고 지정한 것이다.

**4. 고전적인 전략 패턴**

전통적인 방법으로 구현한 전략 패턴을 소개 하려고 한다. 체력치 계산 함수를 나타내는 클래스 계통을 아예 따로 만들고, 실제 체력치 계산 함수는 이 클래스 계통의 가상 멤버 함수로 만드는 것이다.



GameCharacter가 상속 계통의 최상위 클래스이고, EvilBadGuy 및 EyeCandyCaracter는 여기서 갈ㄹ라져 나온 파생 클래스 이며, 한편 HealthCalcFunc는 SlowHealtherLoser 및 FastHealthLoser등을 파생클래스로 거느린 최상위 클래스 라는 것뿐이다. 그리고 GameCharacter 타입을 따르는 모든 객체는 HealthCalcFunc 타입의 객체에 대한 포인터를 포함한다는 이야기이다,.

|  |
| --- |
| class GameCharacter; // 전방 선언  class HealthCalcFunc {  public:  ...  virtual int calc(const GameCharater& gc) const { ... }  ...  };  HealthCalcFunc defaultHealthCalc;  class GameCharater {  public:  explicit GameCharater(HealthCalcFunc \*phcf = &defaultHealthCalc) : pHealthCalc(phcf) { }  int healthValue() const  { return pHealthCalc->calc(\*this);)  ...  private:  HealthCalcFunc \*pHealthCalc;  }; |

이 방법은 “표준적인” 전략 패턴 구현 방법에 친숙한 경우에 빨리 이해할 수 있다는 점에서 매력적이다. 게다가 HealthCalcFunc 클래스 계통에 파생 클래스를 추가함으로써 기존의 체력치 계산 알고리즘을 조정/개조 할 수 있는 가능성을 열어 놓았다는 점도 플러스 이다.

**지금 까지 내용들의 요약**

**1. 비 가상 인터페이스 관용구(NVI) 를 사용합니다. :** 공개 되지 않은 가상 함수를 비가상 public 멤버 함수로 감싸서 호출하는, 템플릿 메서드 패턴의 한 형태이다.

**2. 가상 함수를 함수 포인터 데이터 멤버로 대체 합니다 :** 군더더기 없이 전략 패턴의 핵심만을 보여주는 형태이다.

**3. 가상 함수를 tr1::function 데이터 멤버로 대체하여, 호환되는 시그터처를 가진 함수호출성 개체를 사용할 수 있도록 만듭니다 :** 역시 전략 패턴의 한 형태

**4. 한쪽 클래스 계통에 속해 있는 가상 함수를 다른 쪽 계통에 속해 있는 가상 함수로 대체합니다 :** 전략 패턴의 전통적인 구현 형태

**요약**

-가상 함수 대신에 쓸 수 있는 다른 방법으로 NVI 관용구 및 전략 패턴을 들 수 있습니다. 이중 NVI 관용구는 그 자체가 템플릿 메서드 패턴의 한 예입니다.

- 객체에 필요한 기능을 멤버 함수로부터 클래스 외부의 비멤버 함수로 옮기면, 그 비멤버 함수는 그 클래스의 public 멤버가 아닌 것들을 접근할 수 없다는 단점이 생깁니다.

- tr1::function 객체는 일반화된 함수 포인터처럼 동작합니다. 이 객체는 주어진 대한 시그너처와 호환되는 모든 함수 호출성 개체를 지원합니다.