Utilización de casos de uso en el diseño de aplicaciones en C++.

Víctor Merino

Tool

Acerca de mi

- Programando en C++ desde hace 25 años en el campo de la Ingeniería Civil en Tool
- Actualmente mi trabajo consiste en la integración del diseño con la gestión del proyecto, para poder saber de forma rápida el impacto que cambios en el diseño tienen en el costo y duración del proyecto.
- Miembro del comité nacional de estandarización de C++
- Corresponsable del Subgrupo 4.1 Formatos Interoperables de la Comision BIM para la implantanción del BIM en España.

El problema

- Aplicaciones con clases que tienen cientos de funciones.
- Funcionalidad dispersa entre decenas de objetos. El algoritmo es difícil de saber como es realmente.
- Hay acuerdo sobre que se emplea 10x mas tiempo en leer y entender el código existente que en escribir nuevo código.

Una solución: DCI

Trygve Reenskaug y James Coplien

Los autores de DCI

- Trygve Reenskaug formulo el patrón Modelo Vista Control en 1979 mientras trabajaba en Xerox PARC
- James Coplien fue el primer usuario de C++.
- DCI fue inventado por Trygve Reenskaug. Su formulación actual es principalmente el trabajo de Reenskaug y James Coplien.

Los casos de uso

- La utilización de casos de uso juega un papel fundamental en DCI.
- Los casos de uso han sido usados como un modo de expresar las especificaciones desde hace 30 años. Fueron introducidos por primera vez en OOPSLA 87 aunque no fueron adoptados de una forma generalizada hasta 1992 con la publicación del libro "Object-Oriented Software Engineering—A Use-Case–Driven Approach".
- En DCI los casos de uso tienen que ser "esenciales". En el sentido del libro "Software for Use" de Constantine & Lockwood.
- Simplificado, abstracto, generalizado que captura las intenciones de un usuario de un manera independiente de la implementación y tecnologia.
- Evitar los casos de uso de destrucción masiva con una complejidad innecesaria debida a la burocraticación y que afecta de forma muy negativa a la productividad.

Plantilla de un caso de uso

- Nombre
- Propósito del usuario
- Motivación
- Precondiciones
- Pasos
  - Numero
  - Intención del Actor
  - Reponsabilidad del sistema
- Variaciones
- Postcondiciones

Que añade valor a un caso de uso

- 1. Ayuda al usuario a alcanzar su meta
- 2. Identifica todos los posible factores que pueden conducir al fracaso
- 3. Logra mantener las garantias a pesar de no alcanzarse la meta. Las garantias son las poscondiciones del caso de uso.

La energía requerida va en aumento

- La energía requerida va aumentando de 1 a 3. Esto tiene importantes repercusiones a la hora de programar. Si en el análisis o diseño, no se han identificado los factores que pueden conducir al fracaso, es difícil que se haga al programar. Y muchas veces aparecerán durante la utilización del software, con un costo mucho mas elevado.

E.1: Develop an error-handling strategy early in a design

- Esta es una core guideline la primera del grupo de Error Handling.
- **Reason:** A consistent and complete strategy for handling errors is hard to retrofit into a system.
- Esta etapa del desarrollo es la primera en que se puede diseñar una estrategia de gestión de errores. Esto tiene una repercusión directa en el coste del desarrollo.

Refactorizar

- Es difícil hacer ingeniería inversa, esto es establecer los casos de uso a partir del código, en un sistema que no se ha desarrollado partiendo de los casos de uso porque estarán dispersos por múltiples localizaciones.
- El enfoque de Test Driven Development me parece que es igualmente desarrollar un caso de uso a la vez que se programa.

Datos, Contexto, Interacción

- Los datos corresponden en DCI a los objetos del dominio del problema.
- El contexto es la expresión directa del caso de uso.
- La interacción consiste en los papeles que van a jugar los objetos del dominio del problema para lograr el propósito del caso de uso.

Datos

- Los objetos del dominio del problema se modelan de la forma habitual.
- No tienen mas funciones que las de acceso.
- Estos objetos no son muy inteligentes.
- Son la parte mas estable del sistema. No cambian muy a menudo.
- Representan el estado del sistema.
- No van a crecer hasta tener cientos de funciones, porque estas funciones se inyectan posteriormente durante la ejecución de los casos de uso.

Contexto

- Un contexto representa un caso de uso. En el estará reflejado el escenario mas sencillo, donde nada falla y todo va sobre ruedas. Y cada una de las variaciones que se pueden dar cuando las cosas se tuercen.
- Es el tablero de juego donde se asignan los papeles que van a representar cada uno de los objetos del dominio del problema.
- El algoritmo que vamos a utilizar con sus variaciones estará localizado y podremos leerlo.
- Según vayan apareciendo nuevos casos de uso, si nuestra aplicación tiene éxito y se usa, iremos añadiendo nuevos contextos.
- También pueden aparecer nuevas variaciones en un caso de uso que ya teníamos. Sin embargo los objetos del dominio del problema permanecerán estables.

Funcionalidad de Contexto

- Buscar los objetos participantes
- Asociar estos objetos con los roles que van a jugar
- Comenzar la representación cuando se invoque el método determinado para ello
- Publicar las conexiones con el subsistema Interacción con Usuario para su uso por los objetos rol con métodos.

Interacción

- Representa el comportamiento del sistema.
- Esta compuesto por objetos rol sin estado.
- Dos tipos de objetos rol:
  - Sin funciones.
  - Con funciones.
- Los objetos rol sin funciones hacen de identificadores. Y son interfaces con todos los métodos virtuales puros.
- Los objetos rol con funciones tienen las funciones genéricas que vamos a inyectar a los objetos del dominio del problema para que puedan representar el papel que se les ha asignado en este contexto.

Un contexto

```
12  class Ejemplo_Contexto: public Contexto
13  {
14  public:
15      // Constructores / destructor
16      Ejemplo_Contexto(void);
17      Ejemplo_Contexto(Clase_1, Objeto_Rol_Sin_1*, Objeto_Rol_Sin_2*)
18
19      void hazlo(void);
20
21      Objeto_Rol_Sin_1* papel_1(void) const;
22      Objeto_Rol_Sin_2* papel_2(void) const;
23      Clase_1 valor() const;
24
25  private:
26      void busca_objetos(void);
27
28      Objeto_Rol_Sin_1* papel_1_;
29      Objeto_Rol_Sin_2* papel_2_;
30      Clase_1 valor_;
31  }
```

Implementación de un contexto 1

```
8   Ejemplo_Contexto::Ejemplo_Contexto(void): Contexto()
9   {
10      busca_objetos();
11  }
12
13  Ejemplo_Contexto::Ejemplo_Contexto(Clase_1 un_valor, Objeto_Rol_Sin_1* un_papel, Objeto_Rol_Sin_2* otro_papel):
14      Context()
15  {
16      papel_1_ = un_papel;
17      papel_2_ = otro_papel;
18      valor_ = un_valor;
19  }
20
21  void
22  Ejemplo_Contexto::hazlo(void)
23  {
24      papel_1()->intervencion_1(valor());
25  }
26
27  void
28  Ejemplo_Contexto::busca_objetos(void)
29  {
30      papel_1_ = new Clase_Dominio_Problema_1;
31      papel_2_ = new Clase_Dominio_Problema_2;
32      papel_1_->funcion_1(100.00);
33      papel_2_->funcion_1(500.00);
34      valor_ = Clase_1(30.00);
35  }
```

Implementación de un contexto 2

```
36
37  Objeto_Rol_Sin_1*
38  Ejemplo_Contexto::papel_1(void) const
39  {
40      return papel_1_;
41  }
42
43  Objeto_Rol_Sin_2*
44  Ejemplo_Contexto::papel_2(void) const
45  {
46      return papel_2_;
47  }
48
49  Clase_1
50  Ejemplo_Contexto::valor(void) const
51  {
52      return valor_;
53  }
```

Objeto rol sin funciones

```
4   #include "Clase_1.hpp"
5
6   class Objeto_Rol_Sin_1 {
7   public:
8       virtual void intervencion_1(Clase_1 un_valor) = 0;
9       virtual void intervencion_2(Clase_1 un_valor) = 0;
10      virtual void intervencion_3(void) = 0;
11  };
12
```

Objeto rol con funciones

```
14
15  #define SELF static_cast<ConcreteDerived*>(this)
16  #define ROL_2 ((static_cast<Ejemplo_Contexto*>(Contexto::contexto_actual_)->papel_2()))
17
18
19  using namespace std;
20
21  template <class ConcreteDerived>
22  class Objeto_Rol_Con_1: public Objeto_Rol_Sin_1
23  {
24  public:
25
26      void intervencion_1(Clase_1 un_valor) {
27          if (SELF->valor() > un_valor) {
28              SELF->funcion_1(un_valor);
29              ROL_2->funcion_2(un_valor);
30              SELF->funcion_3("Un texto ...", DateTime(), un_valor);
31              ROL_2->funcion_3("Un texto ...", DateTime(), un_valor);
32          }
33      }
34  public:
35      Objeto_Rol_Con_1(void)  { }
36  };
37
```

Objeto del dominio del problema

```
10  class Clase_Dominio_Problema_1: public Objeto_Rol_Con_1<Clase_Dominio_Problema_1>
11  {
12  public:
13      friend Objeto_Rol_Con_1<Clase_Dominio_Problema_1>;
14
15      Clase_Dominio_Problema_1(void);
16
17      Clase_1 valor(void);
18      void funcion_1(Clase_1);
19      void funcion_2(Clase_1);
20
21  private:
22      Clase_1 valor_;
23  };
24
25  #endif
```

Cuando no usar DCI

- Si el problema se puede abordar mediante funciones de los objetos, no hay un algoritmo en el que haya que utilizar funciones de varios objetos distintos utilizar DCI sería matar moscas a cañonazos.

Bibliografía

- Coplien, James O. and Bjornvig, Gertrud Lean Architecture: for agile software sevelopment John Wiley & Sons 2010
- Constantine, Larry L. and Lockwood, Lucy A.D. Software For Use Addison Wesley 1999
- Jacobson, Ivar et al. Object-Orientd Software Engineering Addison Wesley 1992
- USE-CASE 2.0 The hub of software development. Ivar Jacobson et al. Communications of the ACM May 2016 VOL. 59 No. 5
- www.drdobbs.com/use-cases-of-mass-destruction/184415814
- https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Re-design
- An Empirical Study on Code Comprehension: Data Context Interaction Compared to Classical Object Oriented Valdecantos H.A. et al. 2017 IEEE 25th International Conference on Program Comprehension (ICPC)