doi:10.19306/j.cnki.2095-8110.2017.06.006

一种基于适配器模式的惯导软件设计方法

解 芳,张凌宇,许 静,王 岩,任建辉

(北京自动化控制设备研究所,北京 100074)

摘 要:随着软件任务的不断增多,传统的软件设计开发和维护模式已无法满足日益增长的任务要求,亟待进一步提高软件的复用程度。针对惯导系统软件接口设计,提出了一种基于适配器模式的惯导软件设计方法。该方法采用动多态适配器和静多态适配器技术,提高了软件开发效率,接口的定义灵活可靠,解决了软件开发过程中面临的可扩展性差、可重用性差、难于维护等问题。

关键词:适配器设计模式;惯导软件

中图分类号:TP311 **文献标志码**:A **文章编号**:2095-8110(2017)06-0037-05

An Inertial Navigation Software Design Method Based on the Structural Design Mode

XIE Fang, ZHANG Ling-yu, XU Jing, WANG Yan, REN Jian-hui

(Beijing Institute of Automatic Control Equipment, Beijing 100074, China)

Abstract: With the incessant increase of software task, the traditional development and maintenance mode of software design has been unable to meet the increasing demands of the task. Therefore, the level of software reuse needs to be further improved urgently. For the inertial navigation software interface design, a software design method based on the Adapter design pattern is proposed. The method that adopts dynamic polymorphic adapter and static polymorphic adapter can improve the efficiency of software development. Because of the good flexibility and reliability of interface definition, the method can solve the problems of poor scalability, reusability and maintenance faced in the software development process.

Key words: Adapter design pattern; Inertial navigation software

0 引言

软件业的发展不仅要求软件有更高的生产率和可靠性,而且对软件的可重用性和可维护性也提出了更高的要求。设计模式主要用于解决软件开发过程中重复发生的问题,每一个设计模式都可以被应用于任何系统,因为它集中于一个特定的面向对象设计问题或设计要点,描述了什么时候使用它,以及使用的效果和如何取舍等,是软件设计过程中的设计经验。

设计模式主要分 3 个类型:结构型(Structural),创建型(Creational)和行为型(Behavioral)。其中,结构型模式主要用于如何组合已有的类和对象而获得的结构,一般借鉴封装、代理、多继承等概念,将一个或多个类或对象进行封装以提供统一的外部视图或新的功能。常见的有适配器模式(Adapter)、合式模式(Composite)、桥接模式(Bridge)、装饰模式(Decorator)等。

相对于其他模式,适配器模式主要是为了解决 2 个已有接口之间不匹配的问题。该模式不考虑这些接口是怎样实现的,也不考虑各自可能会如何演化,从而不需要对 2 个独立设计的类中的任意一个

收稿日期:2016-06-27;**修订日期**:2016-07-30

作者简介:解芳(1983-),女,硕士,工程师,主要从事计算机应用与软件工程化方面的研究。

E-mail: xf0202102@163.com

进行重新设计，就能够使它们协同工作。

适配器模式将一个接口转换成另外的接口，使原本因为接口不兼容而不能一起工作的接口实现在一起工作。适配器模式分为类适配器和对象适配器两种方式，其中类适配器使用多重继承对一个接口和另一个接口进行匹配，如图 1 所示；对象适配器依赖于对象组合，如图 2 所示。

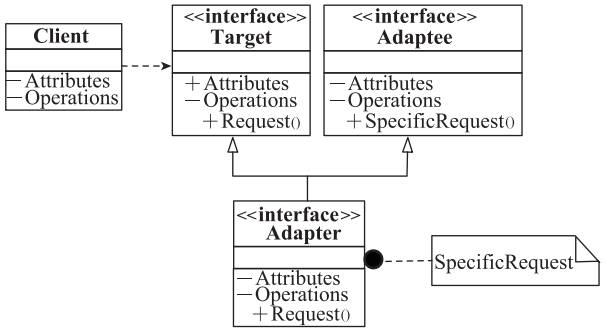

图 1 类适配器说明

Fig. 1 Class adapter illustration

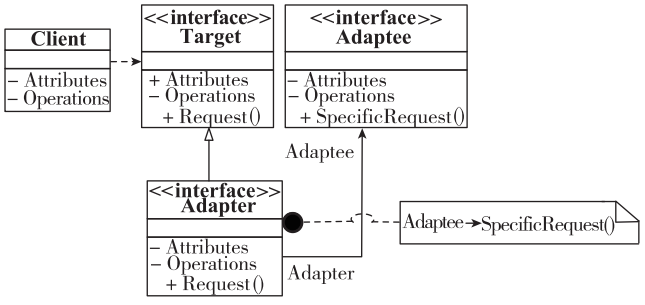

图 2 对象适配器说明

Fig. 2 Object adapter illustration

图 1、图 2 中，Target 为定义 Client 使用的与特定领域相关的接口，是已发布的接口；Client 使用已发布接口的部件与 Target 接口的对象协同；Adaptee 为定义一个已经存在的接口新增加的功能，这个接口需要适配，Adapter 对 Adaptee 的接口与 Target 接口进行适配。Client 在 Adapter 实例上调用一些操作，接着适配器调用 Adaptee 的操作实现这个请求。

1 问题描述

以典型的捷联惯导系统为例，惯导系统软件典型外部接口如图 3 所示。惯导系统与陀螺、控制电路、卫星接收机、测试设备的通信接口为 RS-422 接口、惯导系统与制导计算机的接口为 1553B 接口，惯导系统与温度传感器的接口为 I/O 接口。惯导系统软件如何设计通用接口以满足外部接口的多样性，是惯导系统软件设计亟待解决的问题。

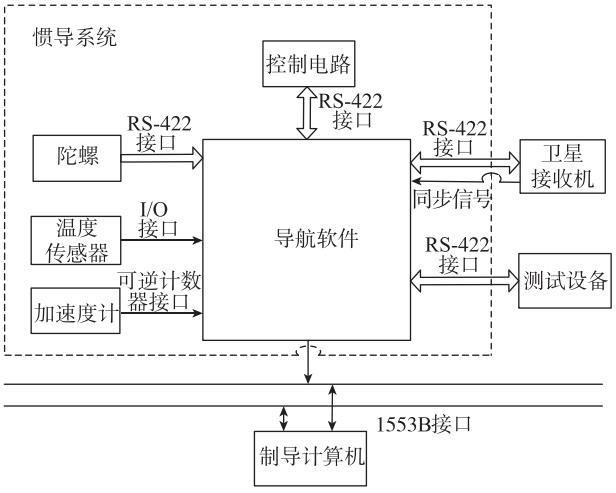

图 3 惯导系统软件外部接口示意图

Fig. 3 Schematic diagram of inertial navigation software external interface

在面向对象技术中，对象通过接口与外部交流，对象接口与其功能实现是分离的，不同对象可以对请求做不同的实现，相同的接口也可以有完全不同的实现。由于惯导系统软件的接口类型较多，硬件设备提供的设备驱动软件接口定义不一致，因此，使用适配器模式适配各种驱动接口，可以隔离硬件驱动的内部变化，使惯导系统软件具备可靠的移植能力，高效的复用性及可扩展性。

2 基于适配器模式的惯导软件设计实例

2.1 适配器设计模式下的惯导软件设计实现

当前惯导系统导航计算机在硬件接口层中，多数项目将相关的硬件协议及软件协议层等通过相关的 FPGA 进行实现，即已经具备相应的各类硬件接口的实现。惯导系统软件只需要完成数据的交互及状态判断等设计。经过分析，只需对软件数据交互接口进行相关分析及设计，其中包括 I/O 接口、内存接口等数据交互。因此通过使用适配器模式设计通用接口适配相应的各类硬件接口，从而完成接口的设计与实现。

下面以实例形式分别实现适配器模式中的类适配器和对象适配器。其中，通过继承和虚函数（动多态，dynamic polymorphism）来实现类适配器。通过模板的泛型标记，可关联不同的特定行为（静多态，static polymorphism）来实现对象适配器。

通过比较说明和分析实现各自的特点，并提出组合实现方案。

2.1.1　类适配器的动多态实现

类适配器的设计思想是对几个相关对象的类型，确定相关的共同功能集，然后在基类中，把这些共同的功能声明为多个虚函数接口。该方法的特点是能够处理异类集合，可执行代码大小相对较小。类适配器的动多态说明图如图 4 所示。

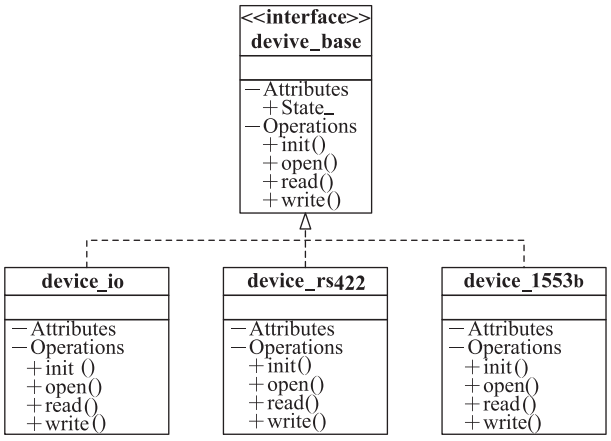

图 4　类适配器的动多态说明

Fig. 4　Dynamic polymorphic illustration of Class adapter

系统接口动多态设计示意图如图 5 所示。

```
struct sp_base_interface
{
    char buf[100];
    int size;

    virtual sp_bool init(void) = 0;
    virtual sp_bool open(void* ) = 0;
    virtual int read(void*, int size) = 0;
    virtual int write(void*, int size) = 0;
    ...
};

struct device_io : public sp_base_interface
{
    ...
    virtual int read(void*, int size)
    {
        ...
        return 0;
    }
    ...
};
struct device_rs422 : public sp_base_interface
{
    ...
    virtual int read(void*, int size)
    {
        ...
        return 0;
    }
    ...
};

struct device_1553b : public sp_base_interface
{
    ...
    virtual int read(void*, int size)
    {
        ...
        return 0;
    }
    ...
};
```

```
void handle(void)
{
    device_io io;
    device_rs422 rs422;
    device_1553b c1553b;
    io.read(io.buf, io.size);
    rs422.read(rs422.buf, rs422.size);
    c1553b.read(c1553b.buf, c1553b.size);

    // 处理异类集合
    sp_buffert_t< sp_base_interface> buf;
    buf.push_back(io);
    buf.push_back(rs422);
    buf.push_back(c1553b);

    for (it = buf.begin(); it != buf.end(); ++it)
        it->read(it->buf, it->size);
}
```

图 5　系统接口动多态设计示意图

Fig. 5　Sketch map of system interface dynamic polymorphic design

2.1.2　对象适配器的静多态实现

模板用于实现多态时，相关函数必须具有相同的名称。虽然不能透明地处理异类集合，但能够在性能和类型安全方面带来显著的好处。该方法的特点是容易实现内建类型的集合，可不需要公共基类来表达接口的共同性，生成的代码效率比较高；可以只提供部分接口的具体实现，不必提供全面的接口实现。

对象适配器的静多态说明图如图 6 所示。

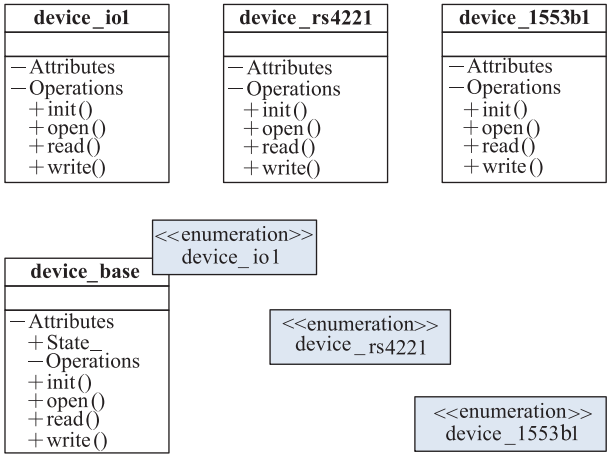

图 6　对象适配器的静多态说明

Fig. 6　Dynamic polymorphic illustration of object adapter

系统接口静多态设计示意图如图 7 所示。

2.1.3　组合这两种实现方式

通过上面两种多态实现接口的方法，可以看出动多态是绑定并且动态的，静多态则是非绑定并且静态的。将这两种方法组合起来可以发挥各自特点。图 8 所示为代码实例。

```
struct device_io1
{
    ...
    int read(void*, int size)
    {
        ...
        return 0;
    }
    ...
};

struct device_rs4221
{
    ...
    int read(void*, int size)
    {
        ...
        return 0;
    }
    ...
};

struct device_1553b1
{
    ...
    int read(void*, int size)
    {
        ...
        return 0;
    }
    ...
};

template <class T>
struct device_base
{
    char buf[100];
    int size;
    ...
    int read(void*buf, int size)
    {
        return s.read(buf, size);
    }
    ...

    T s;
};
void handle(void)
{
    device_base<device_io1> io;
    device_base<device_rs4221> rs422;
    device_base<device_1553b1> c1553b;
    io.read(io.buf, io.size);
    rs422.read(rs422.buf, rs422.size);
    c1553b.read(c1553b.buf, c1553b.size);
    // 不能透明地处理异类集合
}
```

图 7 系统接口静多态设计示意图

Fig. 7 Sketch map of system interface static polymorphic design

两种方法的组合依赖于奇异的递归模板模式(Curiously Recurring Template Pattern, CRTP),该模式代表类实现技术中一种通用模式,即派生类将本身作为模板参数传递给基类。这样可以从公共基类派生出不同种类的接口类,从而可以处理属于异类集合的不同接口对象,同时仍然可以使用模板编写针对某种接口对象的代码。

通过上述实例可见,适配器将一个类的接口转换成用户希望的另外一个接口,使得原本接口不兼容的几个类能一起工作,从而提高了软件的扩展性和通用性。

2.2 效果分析

原来用过程语言实现的接口主要是使用低层次的驱动代码,接口复杂不统一,无法有效复用,但

```
template <class T>
class sp_device_impl : public sp_base_interface
{
public:
    char buf[100];
    int size;
    ...
    int read(void* buffer, int size)
    {
        T* p = static_cast<T*>(this);
        return p->internal_read(buffer, size);
    }
    ...
};

struct device_io2 : public sp_device_impl<device_io2>
{
    ...
    int internal_read(void* buffer, int size)
    {
        ...
        return size;
    }
    ...
};
struct device_rs4222 : public sp_device_impl<device_rs4222>
{
    ...
    int internal_read(void* buffer, int size)
    {
        ...
        return size;
    }
    ...
};

struct device_1553b2 : public sp_device_impl<device_1553b2>
{
    ...
    int internal_read(void* buffer, int size)
    {
        ...
        return size;
    }
    ...
};
void handle(void)
{
    device_io2 io;
    device_rs4222 rs422;
    device_1553b2 c1553b;
    io.read(io.buf, io.size);
    rs422.read(rs422.buf, rs422.size);
    c1553b.read(c1553b.buf, c1553b.size);

    sp_buffert_t< sp_base_interface> buf;
    buf.push_back(io);
    buf.push_back(rs422);
    buf.push_back(c1553b);

    // 处理异类集合
    for (it = buf.begin(); it != buf.end(); ++it)
        it->read(it->buf, it->size);
}
```

图 8 系统接口静多态设计示意图

Fig. 8 Sketch map of system interface static polymorphic design

这种设计能够有效保证接口的效率。通过上述惯导软件设计实例可以看出,按照设计模式中适配器模式进行设计和实现接口,可以有效降低数据交互的复杂性,提高代码编写的效率,保证了接口设计简单,层次清晰,接口灵活可靠,并且可以复用。

当惯导系统的硬件执行速度较快时,可以考虑使用动多态类适配器的方法,实现接口标准化,可扩展性强。如果惯导系统的硬件内存空间较大时,可以通过静多态对象适配器的实现方法,提高软件运行速度,接口的定义灵活可靠。当分析惯导系统

硬件及系统要求可以接受的环境下，结合这两种接口特点组合进行运用，可以充分发挥类适配器与对象适配器的优点，保证软件设计可靠性及软件设计的可复用性。

适配器模式具有很好的复用性和可扩展性。如果功能是已经存在的，只是接口不兼容，那么通过适配器模式可以让这些功能得到更好的复用。在实现适配器功能的时候，可以调用自己开发的功能，从而自然的扩展系统的功能。

3　结论

本文设计了一种基于适配器模式的惯导软件设计方法，适配器模式虽然是一个很常用，也比较简单的模式，但通过适配器模式能够从接口的角度来考虑设计问题，集中体现了面向对象设计的原则之一：针对接口编程，而不是针对实现编程。同时适配器模式也是对“开－闭”原则（即对扩展的开发，对修改的关闭）的具体实现。利用动多态适配器和静多态适配器技术提高了软件运行速度，并提高了软件的可扩展性、可重用性以及维护性。

参考文献

[1] Botha R, Elof J H P. Access control in document-centric workflow systems-an agent-based approach [J]. Computers & Security, 2001, 20(6):525-532.

[2] 刘海岩，锁志海，吕青，等.设计模式及其在软件设计中的应用研究[J].西安交通大学学报，2005，39(10)：1043-1047.

[3] 姚蓓窈，向东游，张华栋.高速串口的软件设计模式研究[J]. 计算机测量与控制，2014，22(7)：2318-2320 .

[4] 伍星，郝惠娣，赵渊.设计模式应用研究[J].机械科学与技术，2003，22 (2)：323-324.

[5] 林舒萍，罗键.设计模式的应用研究[J].计算机工程与设计，2005，26 (11)：2980-2982.

[6] 路立峰，陈平，杜军朝，等.设计模式应用实例[J].计算机技术与发展，2005，15 (2)：134-136.

[7] 肖汉.基于可重用构件的软件开发模式研究[J].微电子学与计算机，2007，24(1)：176-179.

[8] 肖卓宇，何锫.设计模式在系统集成中的应用与研究[J].计算机工程与设计，2007，28(17)：4086-4088.

[9] 唐晓君，刘心松，查小科，等.运用设计模式改进软件设计质量的研究[J].电子科技大学学报，2003，32(2)：169-173.

[10] 李文锦，王康健.源代码中设计模式实例的抽取及验证方法研究[J].计算机应用研究，2012，29(11)：4199-4205.

[11] 童立，马远良.设计模式在基于组件的框架设计中的应用[J].计算机工程与应用，2002，38(17)：123-124.