Software Architecture

- principle and practice

第八讲　软件体系结构描述

本章主要内容：

一、软件体系结构形式化的意义

- 通常，好的结构设计是决定软件系统成功的主要因素，然而，目前许多有应用价值的结构，如管道/过滤器、层次结构、C/S结构等，他们都被认为是一种自然的习惯方法，并应用在特定的形式。因此，软件系统的设计师还没有完全开发出系统结构的共性，制定出设计间的选择原则，从特定领域中研究出一般化的范例，或将自己的设计技能传授他人。
- 要使软件体系结构的设计更具有科学性，一个重要的步骤是就是要有一个合适的形式化基础。这一章我们将介绍软件体系结构的形式化描述（formal specification）。
- 人们已经认识到，对于一个成熟的工程学科，形式模型（formal model）和形式化分析技术(techniques for formal analysis)是基础（cornerstone），只是不同工程学科他们使用的形式不同。形式化方法可以提供精确、抽象的模型，提供基于这些模型的分析技术，作为描述指定工程设计的表示法，也有助于模拟系统行为。因此在软件体系结构领域，它的形式化描述也有很多的用处：
- 1．The architecture of a specific system(指定系统的体系结构)
  - 这种类型的形式化，可以提供从软件结构来设计一个特定的系统，这些形式化方法可以作为功能规格说明书的一个部分，扩充系统结构的非形式的特征，还可以详细分析一个系统。
- An architecture style(软件结构风格)

- 这种类型的形式化，可以阐明一般结构化概念的含义，如，结构化连接、层次结构化的表示、结构化风格，在理想的情况下，这种形式化方法可以提供在结构一级基于推理方法分析系统。

- Formal semantics for architectural description language(结构描述语言的形式语义)
  - 这种类型的形式化，是用语言的方法描述结构，并应用传统语言的技术表示描述语言的语义。

返　回

二、软件体系结构描述的方法

- 软件体系结构描述的方法分
  - 为一般描述方法（或非形式的方法）
  - 形式化描述方法。
- 常用的一般描述方法
  - 主程序和子过程
  - 数据抽象和面向对象设计
  - 层次结构
- 常用的形式化描述方法
  - Category Theory(类属理论)
  - Z Notation(Z标记语言)。
- 1. 主程序和子过程
  - 在结构化范型中，将系统结构映射为主程序和一系列具有调用关系的子过程（或子函数）的集合。主程序充当子过程的调用者，子过程之间又存在着复杂的过程调用关系。过程之间通过参数传入和传出数据。这是软件设计最直接、最基本的结构关系。更复杂的系统设计包含了模块、包、库、程序覆盖等概念。
    - 库提供了系统设计支持的公用的二进制代码，它是设计和调试好的子过程的集合。
    - 包通常是是针对特定应用的类别所提供的公用子过程集合，它可以是源代码或二进制代码形式的。为了支持上运行时组装，通常需要为包提供初始化操作，以维持包的初始状态。

- 模块具有与包类似的结构，但支持上层功能的转换操作，可以看成是具有独立主程序和子过程结构的功能块。
- 程序覆盖是在有限的存储空间条件下，支持功能模块和包对换的机制。

- 优点
  - 是一切软件结构的最本质、最基础的形式，一切软件的结构问题都可以通过在此层次上的代码得追溯；
  - 只要设计得当，代码的效率可以得到最大限度的发挥和提高。
- 缺点
  - 部件的连接关系不明显。直观的部件关系只是过程的调用，但实际支持的连接可以复杂，为把握部件之间的连接关系造成困难。
  - 代码的维护性差。简单的过程调用关系为代码的维护带来困难，特别是数据据结构的变化会引起复杂的关联变化。
  - 代码的重用性差。单纯的过程概念不能反应复杂的软件结构关系，难以成为软件重用的基本单元，不能构成大规模的重用基础。

- 2．数据抽象和面向对象设计

- 数据抽象和面向对象设计是在主程序和子过程设计方法的基础上建立和发展起来的重要的软件描述方法。数据抽象是面向对象设计的理论基础，类和对象是该描述方法的基础粒度。
- 类是数据抽象的载体，由数据成员和操作方法构成，成员的类型和取值范围定义了数据运算的域，方法定义了对数据的运算及其遵循的公理，这样，数据以及对它们的操作就被安然完整地封装在一起而形成了抽象数据类型。
- 对象是类的实例，是软件系统的可运行实体，它全面复制了类的数据组成和操作方法，对象的数据反映了对象的生存状态。对象之间是通过操作方法的调用建立相互作用关系，方法调用采用<对象名>.<方法名>(<参数>)的形式。这就是对象的信息隐藏性和封装性，它保证了行为正确的对象在外界环境不出现意外时永远是正确的。
- 类的继承性是一种重用机制。通过继承类的数据和操作，并加以扩充，可以快速建立（或导出）起新的类。另外，封装性保证

了类或对象作为独立体的完整性。

- 多态性是同一行为名，作用在不同类的对象上时，对应的性质相同但操作细节不同的特性，保证了系统中部件要可替换兼容性。
- 动态链接是在可环境中实现多态性的机制，也是运行代码重用的机制，为运行代码的扩充、升级提供了可能。

3．Category Theory(类属理论)

类属理论是一种表达对象关系的数学语言，最初的研究是Samuel Eilenberg 和 Sanders Maclane提出的。广泛用于描述软件体系结构中的部件和连接器。

返 回

三、Z Notation简介

- Z Notation是由牛津大学（University of Oxford ）的Programming Research Group研究的一种数学语言，基于一阶谓词逻辑（firs-order logic）和集合论（set theory）。使用标准的逻辑操作符（∧、∨、→等）和集合操作符（∩、∪、∈等）以及它们的标准常规定义。
- 使用Z Notation可以描述数学对象的模型，这些对象与程序计算对象相似，这是Z可作为体系结构和软件工程描述的原因。

1．什么是形式规范（formal specifications）

- 形式规范（formal specifications）是用数学表示法（notation）精确地描述信息系统的属性，而不过分注重实现这些属性的方法，也就是说，这些规范描述系统应该做什么而不关心系统如何实现。这种抽象使用得形式规范对开发计算机系统的过程很有意义，因为它们使得系统的问题能够可信赖地回答，没有必要解决大量的程序代码细节。
- 形式规范为客户需求分析、程序实现、系统测试或编写系统用户手册等人员提供一个单一、可靠的资料，因为一个系统的形式规范独立于程序代码，可以在系统开发之前完成。虽然在这些规范在系统设计阶段或客户需求变化时需要改变，但它们是促进对系统达成共识的一个有效方法。

2．Z notation的思想

- Z notation的思想是把描述一个系统的规范分解成若干图表（schema，也称架构或模式），图表（schema）可以再分解更小的图表。在Z中，图表（schema）用于描述系统的静态和动态方面：

- 静态方面（static aspects）包括：
  - 系统的状态；
  - 维持系统从一个状态向另一个状态转换的不变关系（invariant relationships）；
- 动态方面（dynamic aspects）包括：
  - 系统的可能操作；
  - 输入与输出关系；
  - 状态的变化的产生。

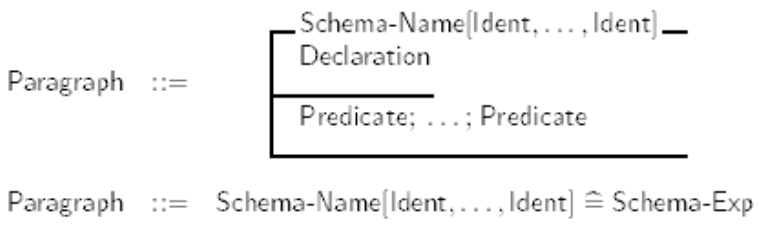
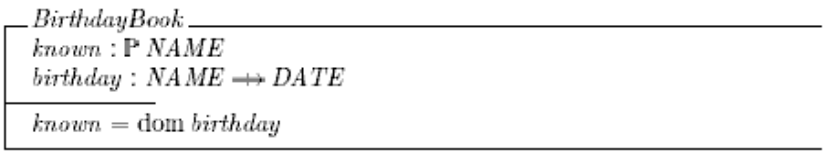


图10-1. Schema definition

-



- *known* is the set of names with birthdays recorded;
- *birthday* is a function which, when applied to certain names, gives the birthdays associated with them.

- **图10-2 schema例子**

3．Declarations

- 声明满足属性的各种变量或变量的取值，在Z中有两种基本的声明方法。
  - Basic-Decl ::= Ident; : : : ; Ident : Expression
  - | Schema-Ref
  - 方法1：
    - X1，X2，...，Xn : E
    - 声明的变量是某集合的一组元素，显式地列出。
  - 方法2：
    - 声明的变量是schema的引用（reference），以schema作为组件。

- 声明变量时，还可以指定是输入或输出，在变量后加上“？”表示该变量从用户输入；在变量后加上“！”表示输出。
- 声明的变量的适用范围由它出现的上下文(context)决定，可以是适合整个规范的全局变量，作为schema的组件，也可以是适合于某个谓词或表达式的局部变量。

4. Schema texts

- 一个schema文本是由声明和或选择的谓词列表。
- Schema-Text ::= Declaration [Predicate; : : : ; Predicate]

5. Predicates

$$
\begin{array}{lll}
\text{Predicate} & ::= & (\text{Predicate}) \\
 & | & \mathit{true} \\
 & | & \mathit{false}
\end{array}
$$

$=, \in$	Equality, membership
$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$	Propositional connectives
$\forall, \exists, \exists_1$	Quantifiers
let	Local definition
Schema-Ref	Schema reference

-

Px	关于X的幂集，即所有以类型X的元素为子集合
$X \leftrightarrow Y$	定义域类型X和值域类型Y之间的所有关系的集合
$X \rightarrowtail Y$	从类型X到类型Y的所有偏函数的集合，一个偏函数是不必在定义域类型的所有元素定义的函数。
$X \rightarrow Y$	从类型X到类型Y的所全函数的集合，一个全函数是在定义域类型的所有元素定义的函数。
$X \rightarrowtail Y$	从类型X到类型Y的所有偏函数、并且其反函数是一个从Y到X的偏函数的集合。也称入函数(injective).
$X \rightarrowtail Y$	从类型X到类型Y的双射函数的集合。双射函数是一一对应的关系。

- ∀ decl|pred1*pred2：它被读作“对在decl中满足谓词pred1的所有变量，都使用谓词pred2为真”；

- ∃ decl|pred1*pred2：它被读作“存在decl中满足谓词pred1的一个或多个变量，使用谓词pred2为真”；

返　回

四、Z－Notation的应用例子- The birthday book

- In our account of the system, we shall need to deal with people's names and with dates. For present purposes, it will not matter what form these names and dates take, so we introduce the set of all names and the set of all dates as basic types of the specification:
- [NAME;DATE]
- 1. state space schema

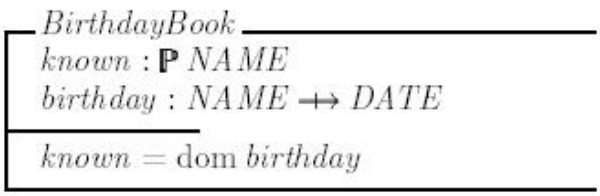


图10.4　state space schema

- Like most schemas, this consists of a part above the central dividing line, in which　ome variables are declared, and a part below the line which gives a relationship　between the values of the variables. In this case we are describing the state space　of a system, and the two variables represent important observations which we　can make of the state:
  - known is the set of names with birthdays recorded;
  - birthday is a function which, when applied to certain names, gives the birthdays associated with them.
- The part of the schema below the line gives a relationship which is true in every state of the system and is maintained by every operation on it: in this case, it says that the set known is the same as the domain of the function birthday - the set of names to which it can be validly applied. This relationship is an invariant of the system.

- Notice that in this description of the state space of the system, we have not been forced to place a limit on the number of birthdays recorded in the birthday book, nor to say that the entries will be stored in a particular order. We have also avoided making a premature decision about the format of names and dates. On the other hand, we have concisely captured the information that each person can have only one birthday, because the variable birthday is a function, and that two people can share the same birthday as in our example. So much for the state space; we can now start on some operations on the system. The fist of these is to add a new birthday, and we describe it with a schema:

◦ operations

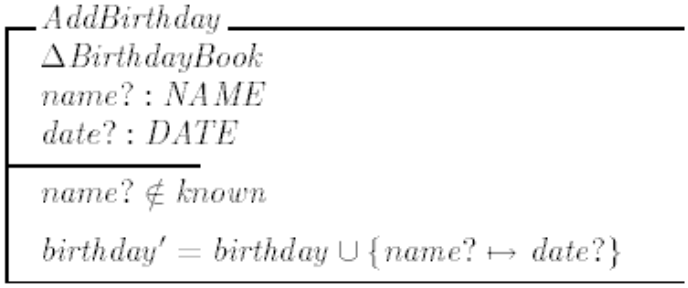


图10.5 operation schema

- The declaration ΔBirthdayBook alerts us to the fact that the schema is describ ing a state change: it introduces four variables known, birthday, known0 and birthday0. The ¯rst two are observations of the state before the change, and thelast two are observations of the state after the change. Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation. Next come the declarations of the two inputs to the operation.By convention, the names of inputs end in a question mark.
- The part of the schema below the line ¯rst of all gives a pre-condition for the success of the operation: the name to be added must not already be one of those known to the system. This is reasonable, since each person can only have one birthday. This specification does not say what happens

if the pre-condition is not satisfied: we shall see later how to extend the specification to say that an error message is to be produced. If the pre-condition is satisfied, however, the second line says that the birthday function is extended to map the new name to the given date.

- We expect that the set of names known to the system will be augmented with the new name:
- Known’ = known ∪ {name?}
- In fact we can prove this from the specifi cation of AddBirthday, using the invariants on the state before and after the operation:

$$
\begin{aligned}
known' &= \mathrm{dom}\ birthday' && [\text{invariant after}]\\
&= \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) && [\text{spec. of } AddBirthday]\\
&= \mathrm{dom}\ birthday \cup \mathrm{dom}\ \{name? \mapsto date?\} && [\text{fact about 'dom'}]\\
&= \mathrm{dom}\ birthday \cup \{name?\} && [\text{fact about 'dom'}]\\
&= known \cup \{name?\}. && [\text{invariant before}]
\end{aligned}
$$

Note:

$$\mathrm{dom}(f \cup g) = (\mathrm{dom}\ f) \cup (\mathrm{dom}\ g)$$
$$\mathrm{dom}\{a \mapsto b\} = \{a\}.$$

图10.6

- Another operation might be to find the birthday of a person known to the system. Again we describe the operation with a schema:

FindBirthday

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

图10.7 operation schema

- This schema illustrates two new notations. The declaration Ξ BirthdayBook indicates that this is an operation in which the state does not change: the values known’ and birthday’ of the observations after the operation are equal

to their values known and birthday beforehand. Including Ξ BirthdayBook above the line has the same efect as including ΔBirthdayBook above the line and the two equations

- known′ = known birthday′ = birthday
- below it. The other notation is the use of a name ending in an exclamation mark for an output: the FindBirthday operation takes a name as input and yields the corresponding birthday as output. The pre-condition for success of the operation is that name is one of the names known to the system; if this is so, the output date! is the value of the birthday function at argument name?.
- The most useful operation on the system is the one to find which people have birthdays on a given date. The operation has one input today?, and one output, cards!, which is a set of names: there may be zero, one, or more people with birthdays on a particular day, to whom birthday cards should be sent.

$$
\begin{array}{|l}
\hline
Remind \\
\Xi BirthdayBook \\
today? : DATE \\
cards! : \mathbb{P}\, NAME \\
\hline
cards! = \{\, n : known \mid birthday(n) = today? \,\} \\
\hline
\end{array}
$$

图10.8

- To finish the specification, we must say what state the system is in when it is first started. This is the initial state of the system, and it also is specified by a schema:

$$
\begin{array}{|l}
\hline
InitBirthdayBook \\
BirthdayBook \\
\hline
known = \varnothing \\
\hline
\end{array}
$$

图10.9

    - This schema describes a birthday book in which the set known is empty: in consequence, the function birthday is empty too.
- What have we achieved in this specification? We have described in the same mathematical framework both the state space of our birthday-book system and the operations which can be performed on it. The data objects which appear in the system were described in terms of mathematical data types such as sets and functions. The description of the state space included an invariant relationship between the parts of the state-information which would not be part of a program implementing the system, but which is vital to understanding it.
- The effects of the operations are described in terms of the relationship which must hold between the input and the output, rather than by giving a recipe to be followed. This is particularly striking in the case of the Remind operation, where we simply documented the conditions under which a name should appear in the output. An implementation would probably have to examine the known names one at a time, printing the ones with today's date as it found them, but this complexity has been avoided in the specification. The implement or is free to use this technique, or any other one, as he or she chooses.
-

返　回