面向最终用户的领域特定语言的研究

Research on a Domain-specific Language for End-user Programming

学校代码：　10248
作者姓名：　郁天宇
学　　号：　1110379023
导　　师：　沈备军
学科专业：　计算机科学与技术
答辩日期：　2014 年 1 月 7 日

上 海 交 通 大 学 软 件 学 院

2013 年 12 月

A Dissertation Submitted to Shanghai Jiao Tong University
for the Degree of Master in Science

Research on a Domain-specific Language for End-user Programming

University Code：	10248
Author：	Tianyu Yu
Student ID：	1110379023
Mentor：	Beijun Shen
Field：	Computer Science and Technology
Date of Oral Defense：	2014.1.7

School of Software

Shanghai Jiao Tong University

Dec. 2013

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：2014年 1月 7日

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在_____年解密后适用本授权书。

本学位论文属于

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：邵天宇　　指导教师签名：沈谷军

日期：2014年 1月 8日　　日期：2014年 1月 8 日

上海交通大学

学位论文原创性声明

本人郑重声明:所呈交的学位论文,是本人在导师的指导下,独立进行研究工作所取得的成果。除文中已经注明引用的内容外,本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体,均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名:

日期:　　年　　月　　日

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在_____年解密后适用本授权书。

本学位论文属于

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月

面向最终用户的领域特定语言

摘要

在传统软件开发过程中，最困难的部分就是开发团队与软件使用人员之间的沟通，这也是软件项目失败或延期的常见原因。因为沟通不到位往往会引起需求不明确或歧义，导致返工。虽然随着软件开发技术的不断发展和完善，沟通环节所带来的弊端不断被减少，但依旧存在风险。同样的风险也存在于系统维护中。如何缓解系统开发和维护中需求不断变更所带来的压力？如何进一步提高系统开发和维护的效率？面对这些问题，本文以高校信息系统为研究领域，提出了一套面向最终用户编程的领域特定语言（DSL），并实现了该语言到 Java 的代码生成器。

本文首先分析并研究了高校信息系统的特点，结合领域特定语言和最终用户编程技术，提出了一套适用于高校信息系统的特定领域语言。高校信息系统因业务繁多，逻辑复杂导致单以单层领域特定语言无法同时兼顾到面向最终用户和语言功能强大两个方面。本文采用分层的思想，从三个层次上设计了高校信息系统的领域特定语言：

第一层，IDSL。这层是面向信息系统的最终用户编程 DSL，本文参考了现有的信息系统 DSL 语言，根据 MVC 架构模式，从三个部分设计了 IDSL：Model、View 和 Logic，分别对应数据模型、界面和后台逻辑。其中，Model 由实体（Entity）组成，并提供增、删、改、查等多种方法，负责相应数据库表的存取；View 通过少量标签支持信息系统中基本的页面定义，并提供增、删、改、查四种基本的业务逻辑操作操作；Logic 支持顺序执行、条件分支和循环结构，并提供了变量申明，方法调用以及跳转至其它 Logic 或 View 的操作。此外 IDSL 不需要用户去定义参数的传递，从而减少了最终用户编程时的难度。

第二层，UIDSL。这层是面向高校信息系统的最终用户编程 DSL，本文在 IDSL 基础上，分析高校信息系统的共性，并以预定义特性的方式进行定义，包括预定义 Enity、预定义 View 和预定义的逻辑操作。这些特性将减少最终用户编程的难度，提高最终用户的开发效率。

第三层，SIDSL。这层是面向高校信息系统选课子领域的最终用户编程 DSL。本文采用领域工程的方法分析系统的共性与可变性，然后统一共性并将可变性参数化以建立 SIDSL 语言。SIDSL 语言采用 XML 方式进行编码，用户只需要对不

同标签选用不同的值作为参数就可以定制选课子领域的不同特征。这种方式使最终用户在更高的抽象层上进行开发，编码也更简洁，效率更高。

接着，根据高校信息系统输入模型复杂，逻辑可变性高这一特点，本文基于重写规则的程序转换技术和 Spoofax 框架，开发了 I2J、U2I 和 S2U 三套工具。

最后，本文选择高校信息系统中的学生选课子系统，针对三层语言进行了三个实验。第一个实验表明 IDSL 编程比 Java 编程的效率要高；第二个实验表明 UIDSL 编程时间小于 IDSL，开发效率更高；第三个实验表明示 SIDSL 的培训时间远小于 UIDSL，开发时间略少于 UIDSL，SIDSL 更加面向最终用户，开发也效率更高。

关键词： 高校信息系统，面向最终用户编程，领域特定语言，代码生成

RESEARCH ON A DOMAIN-SPECIFIC LANGUAGE FOR PROGRAMMING

ABSTRACT

In traditional software development, one of the most difficult parts is the communication between development teams and users, which is a common reason why projects fail or are postponed. Incorrect communication will lead to ambiguous demand and finally rework. While with the continuous development and improvement of software development technology, the risk is reduced but still present. The same risk also influences the maintenance of information systems. How to ease the pressure when the demand for information system changes frequently? How to further improve the efficiency in information systems development and maintenance? Faced with these problems, this paper researches the university information system and put forwards a domain-specific language (DSL) for end-user programming, and develop a code generator to transfer the DSL code to Java code.

This paper firstly analyzes the features of university information system, follows the domain-specific languages and end-user programming techniques and put forwards a domain-specific language for end-user programming which applies to university information system. Because of many businesses and complex business logic in university information system, one single domain-specific language is not enough to take into account both the powerful function and the end-user features. This paper uses three layers to define the domain-specific language for university information system.

The first layer is IDSL. This layer is an end-user DSL for information system. This paper takes the existing DSL for information systems as reference and then defines the IDSL from three parts according to the MVC architectural pattern. These three parts are Model, View and Logic, corresponding to database, page and bussiness logic. Model is composed of Entity and provides some methods such as add, delete, update, and search. It's in charge of the corresponding database table access. View uses a small amount of labels to cover the basic page definition in information system. And it provides four basic business logic operations: add, delete, update and obtain. Logic supports sequential instruction, conditional branch instruction and loop instruction. And it also provides variables declaration, methods invocation and operations used to jump to other Logic and View. In addition IDSL does not require users to define the parameters passing. Thus the difficulty is reduced when end-users are programming.

The second layer is UIDSL. This layer is an end-user programming DSL for university information system. Based on IDSL, this paper analyzes the common features in university information systems and defines some built-in features including built-in Enity, built-in View and built-in logic operations. These features will reduce the difficulty and improve the efficiency when end-users are programming.

The third layer is SIDSL. This layer is an end-user programming DSL for the course selection domain in information system. This paper uses the methods of domain analysis to analyze the commonality and variability. It defines SIDSL by unifying the commonality and parameterizing the variability. SIDSL is an XML style language and users can define different characteristics in course selection domain by choosing different value as parameters for different XML Tag. This approach allows the end users to program at a higher abstraction level. Thus coding is more concise and efficient.

Then this paper uses code generation technology which is based on rewrite rules and Spoofax framework to developed three tools I2J, U2I and S2U according to the complex input model and high logic variability in university information system.

Finally, this paper chooses course selection domain in university information system to finish three experiments for the three languages. The first experiment shows that the efficiency of IDSL is higher than Java. The second experiment shows that the coding time of UIDSL is less than IDSL and the efficiency of UIDSL is higher. The third experiment shows that the training time of SIDSL is much less than UIDSL and the coding time is slightly less than UIDSL. SIDSL is more end-users oriented and efficient.

KEY WORDS: Information system domain, DSL, End-User programming, Code generation

目录

第一章 绪论

1.1 研究背景

最终用户编程已成为软件工程的研究热点之一，因为它具有以下优势：

在传统软件开发过程中，最困难的部分就是开发团队与软件使用人员之间的沟通，这也是软件项目失败或延期的常见原因。因为沟通不到位往往会引起需求不明确或歧义，导致返工。虽然随着软件开发技术的不断发展和完善，沟通环节所带来的弊端不断被减少，但依旧存在风险。最终用户编程提供了一种很好的方法去解决上述问题。最终用户编程是让没有软件开发基础的用户运用他们自己的领域知识去开发软件，最终用户既是开发者也是使用者，他们对自己的需求是最明确，所以不会存在沟通的问题。同时采用最终用户编程所开发的程序是最终用户自己所能理解和看懂的，在软件维护过程中，最终用户也就能了解整个软件的不足，缩短维护周期。

在传统软件开发过程中，供不应求是另一个长期面临的问题。软件工程师的开发速度和生产率达不到用户的需要。由于最终用户较之技术人员多很多，最终用户编程让软件开发能面向非技术人员，受用面更大，软件开发的速度会提高、周期会缩短。

领域特定语言（Domain Specific Language, DSL）是领域工程技术之一，它是针对某一特定领域具有受限制表达性的一种计算机程序设计语言[1]。不同于普通的编程语言，领域特定语言抽象层次更高。针对特定领域，领域特定语言能更简洁、清晰、系统地描述某部分的意图，提高开发效率，使开发过程更加轻松。当其面向最终用户时，最终用户编程将变得更加轻松，自然且富有效率。

因此，研究面向最终用户的领域特定语言，具有良好的研究价值和应用价值。

1.2 国内外研究现状

1.2.1 面向最终用户编程

Lieberman 将最终用户编程定义为：非专业的软件开发人员用户在软件系统应用中可以使用一组方法，如技术和工具，来创建、修改或扩展软件的组件[2]。

目前面向最终用户编程的方法主要有以下几种：

1) 程序合成（Program Synthesis）：典型的方式有两种，一种是示例编程（programming by demonstration），另一种是演示编程（programming by example）。通过例子编程是指用户给计算机执行的原型，通过观察用户输入确定原型；而通过演示来编程是指通过执行一系列计算机需要重复执行的动作，然后把动作通用化的作用于不同的数据集合。Sumit Gulwani[3]为程序合成定义了三个维度：用户目标，搜索空间以及搜索方法。

2) 模型驱动（Model Driven Development）：模型驱动就是注重创造和利用领域模型而非计算概念以提高与系统的兼容性并简化设计过程。

3) 简捷编程（Sloppy Programming）：简捷编程的本质就是允许用户输入一些简单、自然的语言如一些关键字来编程[4-6]，而由计算机去尽可能地理解用户输入的内容，生成相应的代码。

4) 领域特定语言（Domain-Specific Language）：领域特定语言一般面向技术人员。但当其被运用于最终用户编程时，其高效、可跨执行环境、可使用多计算模式的特点[1]将会更有利于最终用户编程。领域特定语言比程序合成适用范围更广，也能创造和利用领域模型来建立语言，同时它还可以配合简捷编程，使得面向最终用户编程更加方便。

1.2.2 领域特定语言

1）领域特定语言的定义

领域特定语言（DSL）是一种为解决特定领域问题而对某个特定领域操作和概念进行抽象的语言[1,7,8]。领域特定语言只是针对某个特定的领域，这点与通用编程语言(General purpose Language)不同，如 Java 既可以适用于网站开发，也可以适用于手机开发。一旦领域特定语言离开了相关领域，它就会变得不适用。但针对某个特定的领域，领域特定语言能很自然地方便地表述问题，也常常比通用编程语言更快地解决问题。

Martin Fowler 在《领域特定语言》一书中认为 [1]：“*Domain-Specific Language (noun): a computer programming language of limited expressiveness focused on a particular domain.*”

他突出了四个关键因素：一是计算机编程语言。它是人类设计出的让计算机执行的命令。故其除了要让人容易理解以外，必须要可执行；二是具有语言的表达能力。语法不能是一个个单独的表达式，必须有一定的组合表达的能力；三是有限的表达能力。它不需要像通用程序语言那样宽的表达能力，只需要它支持某个特定的领域即可；四是关注某个特定领域。只有关注某个特定的领域，有限的表达能力才能发挥最大的作用。第三点与第四点是不能割裂的。

领域特定语言有时候也称为第四代语言，它们常是描述式的，只是说明想要什么，而不像通用编程语言（如 C， Java）那样，要详细写出一步步如何做和实现。Debasish Ghosh[9]总结了两点领域特定语言与其它高级编程语言的差异。一是领域特定语言为用户提供了更高层次的抽象。这使得用户不用去关心诸如特殊的数据结构或低层次实现等细节，而仅仅去着手解决当下的问题即可。二是领域特定语言对于其关注的领域提供了有限的词汇。它不用像通用语言那样为特定的建模领域提供额外的帮助。这两点使得领域特定语言更合适非程序员的领域专家使用。也正是这些特性，通过领域特定语言，能在更高抽象级直观地表达应用问题，能在领域级完成需求确认，实现有效的复用，又能通过工具支持和领域知识复用，使许多从规范说明到可执行代码的转换任务能实现自动化，让应用专长和先进技术能在更广范围内一致使用，避免因开发人员技能水平差异带来大的起伏，从而显著提高开发效率和质量，加快产品开发速度，满足客户多变的需求。

2）领域特定语言的分类

领域特定语言可以分为三类[1,9]：

- 外部领域特定语言(external DSL)
- 内部领域特定语言(internel DSL)
- 语言工作平台(language workbenches)。

外部领域特定语言是完全独立定义的一类新语言。通常外部领域特定语言有一套特有的语法，同时也配有专门的语法解析器和编译器。当然，它用其他语言的语法也是可以的，如 XML。不过由于外部领域特定语言独立于主体开发语言的语言，可能难以与开发的主体语言环境连接，而使问题复杂化。

内部领域特定语言是通用编程语言一种特殊的使用形式。内部领域特定语言无需专门搞一套语法解析器和编译器，而是使用某个通用编程语言，即宿主语言，的语法解析器和编译器即可，如 ruby。这种形式的领域特定语言不仅可以充分利用宿主语言的语法，又相对比较简单，省去了制造语法解析器和编译器的负担，故现在比较流行。不过由于这种 DSL 是搭建于另一种语言之上的，故其效率不高同时可能也会受宿主语言的一些限制。

语言工作平台是一种特殊的定义和建立领域特定语言的平台。语言工作平台不仅是用来确定 DSL 的结构也是作为一个供别人写领域特定语言脚本的自定义编辑环境。Martin Fowler 称语言工作台也许是 DSL 的一个 ‘Killer-App’[10]，它能为老的面向语言编程风格，注入新的元素，诸如：意识软件（Intentional Software）[11]、元程序系统[12]、模型驱动架构（MDA）等。

3）领域特定语言的开发

领域特定语言的开发可以借助于领域工程，即按照领域分析，领域设计，领域实现三个阶段，将一个领域的知识转化成为一组规约、构架和相应的可复用构件。领域分析的主要目的是确定领域范围，分析领域共性和可变性需求，并将结果用易于理解的方式表示出来，形成领域模型。领域设计的目的，就是将软件开发人员在开发同一领域中的系统时逐渐积累起来的经验显式地表达出来，以便在未来的开发中复用。领域实现在领域分析和领域设计的基础上，实现领域中的DSSA 和构件等可复用资产。比较有代表性的领域工程方法包括：FODA（Feature-Oriented Domain Analysis），FORM（Feature-Oriented Reuse Method），FAST（Family-Oriented Abstraction, Specification and Translation），PuLSE（Product Line Software Engineering）等

对于内部 DSL[7,13-16]而言，不需要像外部 DSL 那样去研究语言文法和搞语法分析器，甚至不需要像语言工作台那样开发专用工具。但由于内部 DSL 受制于宿主语言，故选择好的宿主语言成为了相当关键的一步。原则上任何计算机编程语言都可能选作宿主，包括 Java 和 C/C++/C#, 但函数式编程语言，特别是经改进的函数式语言，由于其固有的特点[1,17,18]，成为 DSL 的优选宿主语言。如元编程和闭包的存在会大大简化 DSL 的复杂度。

内部 DSL 常用的一个模式叫流畅接口[1]（Fluent Interface）。Eric Evans 和 Martin Fowler 用这个术语来描述类似语言的 API。在流畅接口模式下，DSL 的可读性会增加。流畅接口的一种核心模式叫方法链 （Method Chaining）。后一个方法调用可以作用在前一个方法的返回值上，从而形成一条方法链。而文字集合（Literal Collection）也比较常用，即将同一模型下的所有属性用放在模型类下。这类似于 setter/getter 方法，但更接近自然语言描述方法，只需属性名称即可。语法树操作也是内部 DSL 常用模式之一，通过对宿主语言的抽象语法树 （AST）操作，对模式的解释生成符合领域要求的代码，在 Groovy 和 Ruby 这是通过它们的库支持完成的。此外 Martin Fowler 也提到利用宿主语言的一些特性如注释（annotation）等进行设计 DSL 的方法。

对于外部 DSL[7,13-16]，语法的自由度会变得更大。但是语法的输入输出需要自己设计。对于输入可以使用 JavaCC 等辅助工具。对于输出，可采用几种方式，一种称为嵌入式翻译 （EmbeddedTranslation），是一个单步的解释执行过程，即边进行语法分析，边收集信息放进一个符号表中，当识别出一个领域模式，即调用相应过程完成一步执行；另一方法是分两步走，先通过语法分析，建立一颗语法树（Tree Construction），再根据语法树用语义模型翻译成代码。

Anneke Kleppe 作为软件语言工程（SLE）的专家，曾系统地讲授如何用元模

型成功设计 DSL[19]，包括语言规范说明的组成、问题领域的概念描述和如何表示，详细的设计策略，如何最大化语言的灵活性，DSL 工具的使用，诸如 Microsoft DSL tools, the Eclipse Modeling Framework, openArchitectureWare 等，以及与通用程序语言的连接，发挥应用开发潜力。

4）国内外现有的领域特定语言

Eelco Visser[20,21]等提出了一套有关 Web 应用的 DSL——WebDSL。Eelco Visser 等认为只有一个比较成熟的领域才合适提出 DSL，而 Web 应用就是比较成熟的领域，有很多技术和框架。Eelco Visser 采用了迭代式的开发过程。对于比较大的领域，可以分解为几个小的范围。先对小的领域进行建模，设计 DSL。然后拓展覆盖范围，修改当前的 DSL 语言，直到语言覆盖整个领域。对于每次的迭代，Eelco Visser 等则是采用了借鉴成熟技术或模板的方法。如 Web 整套逻辑借鉴 MVC 架构将 Web 领域分割成 Mode、Viewer&Controller 三个部分。而对于数据库设计则是借鉴了 Hibernate 的语法进行封装。整个语言是借助于 JSF 框架搭建的。WebDSL 最后会翻译为 JSF 运行。

Zef HEMEL[22]基于 WebDSL 提出了 Mobl，一种智能手机开发应用的 DSL。不过针对手机领域，作者对创建 DSL 的方法做出了改进。由于手机应用的逻辑简化，故作者将 Mobl 的 Controller 部分合并到了 Viewer，由 Viewer 来控制事务逻辑。这也是符合智能手机图形页面操作居多的特性。此外，在设计 DSL 的时候，作者也是考虑到了类似方法链的特性（不过 Mobl 属于外部 DSL）。为 Mobl 设计了两个根概念，窗口（windows）和屏幕（Screen），以形成类方法链。

Martin Bravenboer 和 Eelco Visser[23]提出了一套基于旧 DSL 和 GPL 生成新的 DSL 的方法。混合两种不同语言的核心思想就是使用两种语言的编译器。新 DSL 会由旧 DSL、GPL 和连接语法够成。新 DSL 的输入会由解析器通过连接语法分成旧 DSL 和 GPL。然后分别交由两种语言的编译器进行检查，最后翻译成同一种宿主语言，一般是前面所用到的 GPL。

Ivan Kurtev[24]则提出了通过标准化的元元模型来创建新的 DSL 的一套框架。而元元模型则是使用 MOF 这一标准。这样建立出来的 DSL 能相互进行转化，对于同一个领域也不容易出现异构的描述。

OK 语言[25]则是一套管理信息系统的一套 DSL 语言。语言由表对应实体和页面。页面包含基本的页面操作，包括表单、表格、按钮等。用户不能定义自己的业务逻辑类，只能使用默认的增删改查四种操作。

在软件工程领域，随着需求和技术的不断更新，领域特定语言的种类也越来越多。以下是一些比较常见的领域特定语言：用于描述 Web 页面的 HTML，用

于构造软件系统的 Ant、RAKE、MAKE[9], 用于表达语法的 BNF 范式，语法分析器生成语言 YACC、Bison、ANTLR [9]，用于数据库结构化查询的 SQL，用 Ruby 的行为驱动测试语言的 RSpec, Cucumber[9]，用于描述样式表的 CSS，专用的排版系统 LaTex 等。

1.2.3 研究现状分析小结

无论是面向最终用户编程还是领域特定语言，相关的研究已经持续了较长一段时间，已取得了一定的研究成果，一些领域特定语言更是被投入实际使用，效果明显。本文针对高校信息系统这一领域研究与设计一个面向最终用户的领域特定语言，这在之前没有人做过相关的尝试或研究。

1.3 研究目的和内容

本文以高校信息系统为研究领域，旨在设计了一套面向最终用户编程的领域特定语言，并实现该语言到通用编程语言之间的代码转换工具，以帮助最终用户参与系统的编写，从而提高系统的开发效率，降低成本和风险。

之所以选择高校信息系统为研究领域，是因为本文作者做过高校信息系统的运维，熟悉高校信息系统的业务逻辑。同时信息系统对于高校而言是非常重要的一部分，它能在学生，教师以及管理者中间建立一个收集，分享和处理数据的纽带，也是提高效率的重要手段。但无论是在高校信息系统开发过程中还是运行过程中，需求经常会发生改变，会有大量的开发和维护工作。在传统的软件开发方法中，开发人员常常对高校领域不熟悉，使得需求调研和沟通成本提高。同时需求调研也会有调研得不彻底的情况。这会导致需求不到位，造成大量的返工。而高校信息系统的需求经常变更，这些都使得需求的调研和沟通成为一个瓶颈。对于高校信息系统而言，最终用户的数量远远大于开发人员。如果能让最终用户也能编码，不需要开发人员参与到系统开发工程中，那么需求的调研和沟通成本将会降低，需求不到位的风险也能更好地控制。

为了实现研究目标，本文的研究内容包括：

1) 确定领域范围，分析领域共性和可变性需求，并将结果用易于理解的方式表示出来，形成领域模型。
2) 将领域模型转化为领域特定语言，设计对最终用户的易用性相对高的词汇和语法，设计出领域特定语言。
3) 完成领域特定语言到 Java 的代码生成器的开发，将其生成 SSH 框架的 Java 代码。

4) 实验。选择一个实际子系统，让最终用户和软件工程师分别采用领域特定语言和 Java 语言编程，分析实验结果，验证所研究方法、技术和工具的有效性。

1.4 本文的内容组织

本文共包括五章，各章的内容组织如下：

第一章是本文的绪论部分。包括领域特定语言的概念、面向最终用户编程的综述等，分析本文的研究意义，然后确定本文的研究目标和研究内容。

第二章是高校信息系统领域特定语言的研究与设计。从三个不同层——信息系统层、高校信息系统层以及选课子领域层，设计高校信息系统领域特定语言，包括词法、语法、语义等。

第三章是领域特定语言到 Java 的代码生成器的开发。基于 Spoofax 框架和重写规则设计和实现代码生成器，将领域特定语言的程序转化为目标代码。该章着重阐述代码生成器的整体架构以及具体重写规则的制定过程。

第四章是实验与评估。通过使用领域特定语言进行编程并生成对应的 Java 代码，分析三层高校信息系统领域特定语言的各自优势，以及较 Java 语言的优势。

第五章是总结本文的主要工作成果，对其中的不足进行分析，同时对将来的工作进行展望。

第二章 高校信息系统领域特定语言的研究与设计

2.1 三层 DSL 语言

本章将研究和设计高校信息系统领域特定语言，定义其语法和语义。由于高校信息系统业务繁多，逻辑复杂，单纯定义一层领域特定语言无法使得语言简练且面向最终用户。为了使语言更加灵活多变，功能强大，本文提出了基于分层的思想来定义领域特定语言：

1）信息系统层——IDSL

第一层，针对信息系统领域的 Web 应用系统，设计一个最终用户编程语言 IDSL。直接设计高校信息系统层的领域特定语言比较困难，但若从信息系统层出发设计，再在这层基础上添加特性就形成高校信息系统的领域特定语言的话，就会相对比较容易，而且有一些语言如 WebDSL，OK 作为参考。

2）高校信息系统层——UIDSL

第二层，在 IDSL 的基础上，考虑高校信息系统领域的特点，提取了高校信息系统中特有的表、页面、操作等，并预先定义它们，形成 UIDSL 语言。

3）选课子领域层——SIDSL

第三层，针对高校信息系统系统的一个子领域——选课领域，设计 SIDSL。UIDSL 在 IDSL 的基础上添加了许多预定义的特性，使得语言更加强大、方便。但对于最终用户而言，当遇上比较复杂的业务逻辑时，UIDSL 这套语言的抽象层次还比较低，粒度还比较小，编程还是有一定难度的。为了提高最终用户使用 UIDSL 的便捷性，本文又针对子领域，设计出了更高抽象层次的 DSL——SIDSL。SIDSL 不同于 IDSL 和 UIDSL，它支持可变性的配置或定制，将可变性参数化，进一步方便最终用户编程。

2.2 第一层 IDSL

为了支持最终用户编程，设计一个比普通编程语言更简洁更易学的领域特定语言是首要的任务是，它必须贴近最终用户，符合它们的使用习惯。本节研究并设计了一个支持最终用户编程的信息系统 DSL——IDSL，让领域专家或最终用户，能够通过较短周期的培训，掌握 IDSL 的使用，从而能够开发和维护信息系统。这就要求 IDSL 的设计必须满足以下要求[1]：

1) 相比 Java、C++等普通编程语言，简单易学，有更高的开发效率

2) 在特定的领域有较强的表达能力

3) 可维护性高，可扩展型强

2.2.1 IDSL 语言的设计方法

本文采用了三个步骤来设计 IDSL。

1) 选择合适的架构模式

在 IDSL 的设计过程中，本文并不是从头开始开发框架及其 DSL，而是从现有的应用开始，总结信息系统的特点和常用的开发方法，决定合适的架构模式。本文在研究了多个新信息系统的需求以及遗留信息系统的代码后，选择 MVC（Model, View, Controller）架构模式作为 IDSL 的语言框架，并选择支持 MVC 的 SSH 框架作为目标代码的编程框架。

2) 参考现有相关 DSL 的研究成果，分析现有的信息系统代码，在选定架构的基础上进行抽象，形成语言

本文参考了现有的 Web 领域特定语言——WebDSL 和管理信息系统领域特定语言——OK，分析现有信息系统代码，进行抽象，定义出 IDSL 语言。IDSL 按照 MVC 模式，从 Model、View 和 Control 三方面展开。

2.1）Model 的抽象

根据 SSH 框架，整个工程应该有对应的 Bean，它对应于后台数据库表，数据表的一些简单操作放在 Bean DAO 中。WebDSL 中有 Entity 的概念，对应了数据库表以及相关操作。因此，IDSL 把 Model 定义为多个 Entity 及其关系组成。为了 IDSL 语言的简洁性，所有 set/get 都默认定，而且每个 Entity 都有默认的增、删、改、查四种操作。

2.2）View 的抽象

SSH 框架中用户点击页面调用对应的 StrutsAction 来处理相关用户的请求，OK 语言中有 View 的概念，它将页面和对应的后台操作结合在一起定义。因此，IDSL 把 View 定义为多个 Page，及其和后台 Action 的关系。根据现有的信息系统代码，Page 表现形式主要是表单、列表或者是信息显示，对此 IDSL 抽象出了这三种形式，形成了三种风格的标签。为了面向最终用户，IDSL 舍弃了 Page 和后台 Action 之间的参数传递，改为全局参数，同时定义了默认的增、删、改、查四个操作。用户只要去调用而无需自己去重写这四种操作。

2.3）Logic 的抽象

WebDSL 和 OK 中都没有独立的 Logic 部分，但按 MVC 模式这在信息系统中不可或缺。根据现有信息系统代码，结合 SSH 框架，本文将 IDSL 中的 Logic 设计成多个 Action，并设计了顺序、分支、循环的语句结构。

3) 验证

通过实际信息系统的开发对 IDSL 的有效性与实用性进行验证。发现不足，回到第三步。

在设计 IDSL 时，本文采用自上而下的方法，首先设计了基本的 DSL 片段，完成了 Model 模块的语法设计，验证了 IDSL 在数据模型上的表达能力和转换到 Java 代码的可行性；然后设计了简单的表达式的 IDSL 表达；接着研究基于 Eclipse 的转换工具的设计和开发，以方便实验。在此基础上再进行其他语言要素的扩展。自上而下的方法倾向于快速建立一个完整且自包含的模型，具有更长远的考虑，有助于保证结构的一致性。但为了避免模型复杂，难以实现，在设计的过程中，IDSL 同时采用了渐进的方式，避免早期投入过大风险，并定期设计实验检查转换的可行性。

2.2.2 IDSL 语言的总体结构

根据 MVC 框架，语言由三个部分组成: Model、Logic 和 View。Model 用于刻画数据的存储和访问，Logic 用于刻画应用的后台逻辑，View 用于刻画应用的界面。

MainPage 是整个 IDSL 程序的接入口，每个 IDSL 程序中都必须包含，相当于 Java 中的 Main 函数。所不同的是 MainPage 的参数 View，最终用户只要这样调用即可：MainPage(basicPage)。

2.2.3~2.2.5 小节是这三部分语法的描述，并以选课子系统为例来说明。本文选取了一个高校信息系统中比较简单的学生选课子系统需求。用户是学生，学生点击登录。当身份验证成功，学生跳转到基本页面。学生点击选课，则进入选课页面，学生可以查找自己同时要选的课程并进行选课。同时页面也能显示学生所选的课程，同时也能让学生退课。显示的课程信息包括学生学号、学生导师、学生学院等。

2.2.3 IDSL 语言的 Model

IDSL 语言中 Model 由多个 Entity 组成，下面是 Entity 的语法定义：

```
Entity Id{
    attribute[]:type|attribute:type[]|attribute:type
    function(para*):type?{element*}
}
```

```
element={
    Exp|
    If EXP
    then Statement
    else Statement
    end|
    repeat Statement|EXP until Statement|EXP|
    for()|
    return Exp
}
```

Entity 是关键字，后面跟着 Entity 的名字和具体定义。Entity 的名字命名类似于驼峰命名法，但首字母必须要大写。每个 Entity 在数据库中就有一个对应的表，用于长久存放数据。attribute 表示 Entity 的属性。属性与数据库表中的字段是一一对应的。type 表示属性的类型，可以是任意的 Entity，也可以是默认的基本类型。 []表示数组。根据[]的位置，可以表示一对多或者多对一的关系。Entity 并没有提供多对多的关系，因为多对多的关系可以通过转化为其他的关系来表示。为了保证数据库表中主键的唯一性，每个 Entity 都会有一个自增长的 ID 作为默认的主键。function 表示 Entity 的相关操作，可以有传入的参数 para，也可以有返回类型 type，如果不加返回类型则默认无返回值。element 则是方法的定义，它可以定义参数，也可以调用方法，同时也可以定义条件跳转和循环跳转。if…then…else fi 表示的是条件跳转语句。if 后面跟了条件判断语句，如果条件为真则执行 then 后面的语句否则就执行 else 后面的语句。repeat…until 是循环执行语句。反复执行 repeat 后面的语句知道 until 中的条件判断为真。for(…) …rof 也是循环执行语句。for 的语法和 Java 中的一致。for 后面的（）内跟的是条件判断语句，它首先会初始化一个赋值语句, 它用来给循环控制变量赋初值；条件表达式是一个关系表达式, 它决定什么时候退出循环；增量定义循环控制变量每循环一次后按什么方式变化。这三个部分之间用";"分开对于每个 Entity 的属性，所有属性都有默认的 set 和 get 方法。return 表示的是返回值，可以是任意类型。同时每个 Entiy 也提供以下四种方法：searchByProperty(var:type): List<Entity>，search(Restriction): List<Entity>，save，delete。第一个方法是根据某个属性来查看相应的结果。如 student 有 name 的属性，则查询方法可以表示为 searchByName(“张三”)。整个方法就会去查找数据库中表名是 student 的 name 该方法返回的内容是 List。而 save 是保存方法，delete 是删除方法。如有一个 Entity Student 的实例 student，要将它保存到数据库中，可以调用 student.save()；进行删除，则可以调用 student.delete()。

每个 Entity 在软件中就有一个对应的同名的 BO 类(首字母小写表示，如

Entity Student，则 BO 类为 student)，这个类具有完全相同的属性和方法。

同时每个 Entity 可以通过初始化（如 Student student{ studentId := "1110379023" }）来生产 Entity 类，也可以通过方法返回来生产，这个类也具有完全相同的属性和方法。如果和 BO 同名，则会覆盖 BO 类。Page 的 datagird 或者是 form 提交时，会将对应的信息填写到对应的 BO 类中。BO 类有一个默认的方法是 expire，用于将 Session 中的类注销，以释放 Session 中某个类的信息。

下面的代码显示了选课系统中的 Entity 部分的定义：

```
Entity User{
    userId:String
    passwd:String
    role:Role
    Login(id:String, passwd:String) :Bool{
        List<User> l = User.search(RES(User.id==id AND User.passwd ==
passwd))
        If NOT l.empty()
        then
            return true
        else
            return false
        end
    }
    Logout(){
        user.expire()
    }
}
Entity Role{
    type:String
}
Entity Tutor{
    tutorId: String
    name: String
    school:School
}
Entity Student{
    studentId: String
    name: String
    school: School
    tutor:Tutor
    type:Type
    date:Date
}
```

```
Entity Course{
    courseId: String
    courseName: String
    type: String
    school:School
    tutor:Tutor
}
Entity School{
    schoolId: String
    name: String
}
Entity Schedule{
    student:Student
    course:Course
    season:Date
}
Entity CourseType{
    type:String
    name:String
    score:String
}
Entity Type{
    typeId:String
    typeName:String
}
```

在 Entity User 的定义中，Entity 是关键字，User 是 Entity 名。userId、passwd 等都是它的属性，每个属性都有它的类型，userId 和 passwd 的类型都是 String，role 的类型是 Role，这不是一个基本类型，故它是一个用户自定义的 Entity 类型。Login 和 Logout 是两个方法，Login 传入的参数是 id 和 passwd，返回类型是 Bool。Logout 没有传入的参数，返回类似是默认的 void。Login 有 element 定义使用了条件分支，还使用了 search 这个默认方法，RES 是 restriction，参见 2.2.6。Login 若在 User 表中找到符合要求的就返回 true，否则是 false。

2.2.4 IDSL 语言的 View

IDSL 语言中 View 由多个 Page 组成，下面是 Page 的语法定义：

```
Page pageNamePage(){
    title={message:String}
    pageFrame?
    pageElement*
}
```

Page 是关键字，Page 的名字必须是以 Page 结尾的，如 basicInfoPage。Page

是用来显示页面，同时也包含了一些相关页面的逻辑。关键字 title 的作用类似于 HTMl 中的 title 标签，它是用来显示一个页面的标题信息。标题信息是用{message:String}中的 message 来显示的，它是一个 String 类型的变量。String 类型是预定义的基本变量，和 Java 中的 String 变量一致。pageFrame 表示的是一个页面的框架，即页面布局。pageElement 表示的是页面内容。pageElement 是由 datagrid，form 和 infogrid 组成的。它们稍后会定义，下面是 pageFrame 的语法定义：

```
pageFrame nameFrame(){
    header=[Id]{
        pic={
            Item{imgParameter}*
        }
        link={
            Item{message:String[:page:Page|action:Action]
                [#message:String :page:Page|action:Action]
            }*
        }
    }
    leftBar=[Id]{
        Item{message:String[:page:Page|action:Action]
            [#message:String :page:Page|action:Action]
        }*
    }
    footer=[Id]{
        pic={
            Item{imgParameter}*
        }
    }
}
```

pageFrame 的名字必须由 Frame 结尾，如 basicFrame。pageFrame 的基本布局是微软提出的一套便于人们使用的布局，即分为 header，leftbar 和 footer 三个部分。

关键字 header 表示的是页面顶部的布局。而 header 包括 pic 和 link 两个部分。关键字 pic 表示的是顶部布局中的图片。如果有多个图片，那么图片之间则以百叶窗的形式进行切换。pic 中一个 item 就表示一个图片，imgParamter 表示的是图片的 URL 地址。此 URL 地址采用相对地址，存放图片的位置则是在工程下的 img 文件夹中。关键字 link 则是表示在图片下方的超链接。一个 Item 表示一系列连接，一行一个链接。一个 Item 项的结构就如同树一样，其中第一行的链接表示根超链接，其余的链接都是它的子孙链接。所有的 Item 项都必须包含

一个根超链接。它们是会在页面初始化的时候显示，而其余的子孙超链接则会折叠在自己的父超链接下不显示出来。每一个超链接的语法都形如 message:String[:page:Page|action:Action]，其中 message 表示超链接的名字，Page 或 Action 则表示超链接跳转的目的地址。一个超链接可以跳转到相关页面，也可以跳转到相关业务逻辑。除了根超链接，其余的链接必须加若干“#”符号，表示低一级的超链接。在 message 前加一个#符号，则表示低一级的超链接；两个#，则表示低两级，以此类推。在初始状态下，页面只会显示 message 前没有#符号的链接。其余超链接会折叠起来放在其上一级的超链接下。如果某个超链接下面有子超链接的话，移到相应的链接上，则子链接就会显示出来，若在移到某个子链接上，则子链接的子链接就会显示，如同 windows 的多级菜单一样。

关键字 leftBar 表示的是页面左侧的布局。leftbar 中的 Item 部分语法和 header 中 link 域的 Item 一致，但显示的效果和 header 中 link 域不同。则此链接的显示效果如图 2-1 所示的标志。“+”表示子超链接没展开，“-”表示子超链接已经展开。

图 2-1 leftbar 的效果
Fig. 2-1 Page Display of Leftbar

关键字 footer 表示的是页面底部的布局。它就由图片组成，其语法和 header 中的 pic 域的语法一致。

最终用户可以对 pageFrame 中的 header，leftbar 和 footer 三个部分进行命名。如果最终用户没有命名的话，代码生成器则会自动生成默认的名字，如 header1，leftbar1 等。默认名字的命名方式就是关键字加序号。它们会当做变量来存储，最终用户可以在不同地方访问到他们。为了避免名字重复，每个变量都有不同的域。具体的变量存储，访问会在 2.1.4 节中阐述。

pageElement 包含三种不同风格的表现形式：datagrid，form 和 infogrid。

1) datagrid

datagrid 主要是对多条记录进行显示和增删改查的，下面是 datagrid 的语法定义：

```
datagrid=[Id]{
    headerRow={message:String}
    data={
        (message:String(<:|:>|<>)property: Basic type)*
            Where = {…}
        }
    column={
        (*|message:String:controllerType)*
    }
    action={
        (message:String=actionType[->action:Action])*
    }
    search={
        (message:String[:controllerType])*
    }
    additional={
        order: message:String:[desc]
        partition:number:int
        setting:vertical| horizontal
        submit:radiobox|checkbox
    }
}
```

和 pageFrame 中的 header，leftbar 和 footer 三个部分一样，最终用户可以对每个 datagrid 进行命名。其默认的名字是 datagrid 加序列号。一个 datagrid 由 headerRow、data、column、action、search 和 additional 几部分组成。

headerRow 类似于 HTML 中的 label 标签，用于对整个 datagrid 的功能等进行描述。最终用户输入的 message 信息会以居中加粗大字的形式显示在整个 datagrid 内容的开头。

data 是用于和 Entity 进行绑定的，一行绑定一个 Entity 的属性。其基本的绑定语法形如 message:String(<:|:>|<>)property: Basic type。message 表示绑定时每个属性的所对应的变量的名字，这些变量在当前所在的 datagrid 中均可以被访问和使用。Property 对应的是 Entity 的属性，但用来绑定的属性的类型必须是基本类型，如 Entity Student 中有一个属性是 name，其类型是 string，则它可以通过 Student.name 来表示。如果 Entity 中某个属性的类型是另一个 Entity, 则必须通过进一步的“.”操作来访问这个内部 Entity 的属性，如 Student.tutor.name 就是访问 Student 下 tutor 的 name 属性。<:|:>|<>这三个符号是绑定符号。<:表示只读（readonly），表示显示数据库信息，但不允许使用者通过这些变量来修改 Entity 中的属性从而修改数据库信息。只读操作会被代码生成器解析以 label 标签来显

示，即不能修改。:>表示只写（writeonly），允许使用者通过这些变量来修改 Entity 中的属性以改变数据库信息，但在任何时候都不提供对数据库信息显示的功能。只写操作会被代码生成器解析以 text 标签来显示，但不会在 text 域中显示数据库信息。<>表示读写（R&W），表示会显示数据库信息，同时允许使用者对其进行修改以改变数据库信息。读写操作会被代码生成器解析以 text 标签来显示，同时会在 text 域中显示数据库信息。在 data 中还有一个关键字 Where，它是用于对绑定的信息进行筛选过滤的。其语法和 SQL 中 where 语句的语法类似。data 域的功能相当于将数据库中的信息通过 select…from…where 语句或者 select…from 语句提取出来，然后将不同的列和不同的变量进行绑定。其中 select 出来的字段就是属性名，而 from 的表则是绑定时的 Entity 名和作为 Entity 属性的 Entity 名。多表之间的关联查询会根据 Entity 中一对多或多对一的关系转化为 Hibernate 表述来关联。

column 用于显示 data 中筛选出来的数据并为每一行（列）数据提供行（列）头。column 中每一行就能显示一行（列）data 中筛选出的数据，并提供行（列）头。column 中显示的行（列）数目必须和 data 中数据绑定的数目是相同的，而且顺序和 data 中数据绑定的顺序是一致的，无需最终用户指明。同时最终用户在 additional 中可以指明 column 中的数据显示是竖排还是横排。每行语法形如 *|message:String:controllerType。data 的作用只是从数据库中筛选出来并做了绑定。但当显示数据时，必须在页面上明确每一行（列）数据的含义。column 中 message 就提供了这一功能。controllerType 则用来提供数据显示时所需的控件类型。最终用户不选则表示普通的 text 框。controllerType 包括 radiobox，checkbox，passwordbox 三种。radiobox 是单选框，checkbox 是多选框，passwordbox 是密码框。当用*时则表示对应的一行（列）不显示。

action 用以提供业务逻辑操作。一般的，信息系统业务逻辑操作包括增、删、改、查。因此 action 提供的默认的业务逻辑操作有 add、delete、update 和 obtain。其中 add 会被代码生成器转化为 insert 语句，update 则是 update 语句。由于数据库表中的主键是个自增长的流水号，最终用户不用关心，故任何数据采用 add 操作都不会报错。而 update 会根据使用者的输入，在数据库中找完全一样的数据记录，然后修改此记录。如果数据中存在两条以上完全一样的数据，则 update 操作只会修改第一条记录，以免由于使用者的疏忽而引起一些数据的丢失；如果没有完全一样的数据记录则不会去修改数据库。delete 操作会根据使用者的输入查找完全一样的数据记录然后删除这条记录。如果数据库中存在两条以上完全一样的数据，则 delete 操作只会删除第一条记录，以免由于使用者的疏忽而引起一些数据的丢失；如果没有完全一样的数据记录则不会去修改数据库。obtain 操作会根据使用者的输入查找完全一致的数据，如果找到则会返回 true，如果没有找到则

会返回 false。除了这四个业务逻辑操作以外，action 也提供了另外两个业务逻辑操作：ok 和 reset。ok 是用了调用最终用户自己定义的 action 操作，如 ok->selectCourse 就是调用了 selectCourse 这个最终用户自己定义的业务逻辑。reset 操作是重置页面信息用的。action 每一行的语法形如 message:String=actionType[->action:Action]。action 的每一行都会被代码生成器转化为一个 button。一旦点击了这个按钮，相应的业务逻辑就会被调用。message 表示显示在按钮上的内容，actionType 是六个定义的业务逻辑之一。若 actionType 是 ok 时，必须跟->action:Action 来调用自己定义的业务逻辑。

search 用以提供查询时的条件过滤功能。datagrid 中所有显示的数据记录会在 data 中绑定，而 search 就是对这些绑定的数据提供过滤功能。换而言之，search 中提供条件过滤的字段必须是在页面中显示出来的，即在 column 中显示的字段（非*）。search 的语法很简单，每一行提供一个条件过滤的字段。search 每行的语法形如 message:String[:controllerType]。message 字段必须包含在 column 中的 message 字段中。controllerType 则用来提供控件类型。最终用户不选则表示普通的 text 框。controllerType 包括 radiobox，checkbox 和 list。radiobox 是单选框，checkbox 是多选框，list 则表示下拉列表。页面显示时，一行一个 label 标签和 text 框。label 标签显示 message 字段，用以明确条件过滤的字段名；而 text 框则用于输入每个条件过滤的具体内容。当 controllerType 被选为 radiobox 或 checkbox 时，这行的条件过滤字段的输入框会变成一组单选框或多选框。而选框的内容则是数据库表中相应字段的所有的值。当 controllerType 被选为 list 时，这行的条件过滤字段的输入框就会变成一个下拉列表，而下拉列表中的值则是数据库表中相应字段的所有的值。

additional 提供一些辅助功能。关键字 order 会根据 message 字段提供排序功能。而 message 必须是在 column 中出现过的且 controllerType 是必须是默认的即 text 类型的。关键字 partition 提供分页功能，number 表示每页显示的数据数，它是 int 类型。关键字 setting 提供横排或竖排功能，vertical 是竖排 horizontal 是横排。关键字 submit 提供页面信息提交时的提交方式。radiobox 是单选，checkbox 是多选。当没有关键字 submit 时，每行（列）数据结尾都会有 action 中所提供的所有操作，一旦点击这些按钮相应的操作就会作用于这行（列）。而当有关键字 submit 时，所有 action 中提供的操作会出现在整个 datagrid 的最底部，而每行（列）数据的开头会有单选框或多选框。当某一行或几行被选择好，点击相关按钮，相关的操作就会作用于对应的数据行。

2) form

form 提供了类似 form 表单的功能。这些功能 datagrid 基本都能提供，但为

了使用方便而提供了相应的功能。下面是 form 的语法：

```
form=[Id]{
    data={
        (message:String(<:|:>|<>)property:Basic type )*
            Where = {…}
    }
    column={
        (message:String[:controllerType])*
    }
    action={
        (message:String=actionType[->action:Action])*
    }
}
```

和 datagrid 类似，最终用户可以对每个 form 进行命名。其默认的名字是 form 加序列号。form 包含三个部分：data，column 和 action。它们的语法和 datagrid 的是一样的。一般而言表单的功能让使用者输入信息，点击提交后做相应的业务操作。故而一系列的业务逻辑操作是针对一条数据记录而不是多条。而且往往表单提交是完全有用户输入，然后提交的。之所以在 data 中提供了只读操作是为了能让最终用户给使用者提供一些有用的信息，包括警告信息、建议等。而读写操作是方便最终用户在使用者多次提交表单时显示历史信息用的。为了和传统中表单的使用方法一致，form 在数据筛选时如果出现多条记录，则只会选择第一条记录进行数据绑定和显示。同时所有信息都是竖排的，而且 action 中的所有业务操作都会显示在整个 form 内容的底部。与 datagrid 不同的是，form 提供了验证码的功能。页面上验证码显示在业务逻辑操作的那行的最后。所有业务逻辑操作除了 reset 在提交之前都需填写对验证码。一旦验证码填写错误，页面会有提示，同时业务逻辑操作也不会执行。

3) infogrid

infogrid 提供了对于单一 Entity 内容的显示功能。在信息系统中，显示个人信息或者交易信息之类的功能是比较常见的。它们的特点是数据的显示不需要多表之间复杂的 join，通过外键就能得到数据。换而言之，访问单张表就能显示大部分信息，而其他信息能通过简单的关系如外键就能获取。这些功能 datagrid 均能提供，但使用 infogrid 更为方便。下面是 infogrid 的语法：

```
infogrid=[Id]{
    photo{photoURL:String}
    data{
        #(message:String[(<:|:>|<>) property: Basic type])*
        Where={…}
```

```
    }
    action={
        (message:String=actionType[->action:Action])*
    }
}
```

和 datagrid 类似，最终用户可以对每个 infogrid 进行命名。其默认的名字是 infogrid 加序列号 infogrid 包括三个部分：photo，data 和 action。关键字 photo 用于显示照片，这主要是为了方便最终用户在显示个人信息时提供照片信息。photoURL 表示的是照片的 URL 地址。此 URL 地址采用相对地址，存放照片的位置则是在工程下的 photo 文件夹中。data 部分和 datagrid 中的 data 部分语法类似。唯一不同的是 infogrid 下的每行前面可以加“#”符号。每个#代表了一行的开始。这样最终用户就能根据每项内容的多少来进行页面的布局。个人信息或者交易信息之类基本上是一次显示一条记录，故 infogrid 被设计成只显示一条记录的，如果在数据筛选时如果出现多条记录，infogrid 则会只显示第一条记录。

下面是选课系统中 View 的定义：

```
MainPage(loginPage)

Page loginPage(){
    title={“登录”}
    form=aform{
        data={
            “用户名” <: User.id
            “密码” <: User.Passwd
        }
        column={
            “用户名” “密码”:password
        }
        action={
            “登陆”:ok-> loginAction
            “重置”:reset
        }
    }
}

Page basicPage(){
    title={“个人信息”}
    basicFrame()
    infogrid=basicInfo{
        #”学生学号” <: Student.Id
        #”学生导师” <: Student.tutor.name
```

```
        #"学生学院" <: Student.school.name
    }
}
//显示课程
Page eduplanPage(){
    title={"显示课程"}
    basicFrame()
     datagrid={
        headerRow={"可选课程"}
        data={
            "课程名字" <: Course.courseName
            "课程代码" <: Course.courseId :> Schedule.course.courseId
            "课程类型" <: Course.courseType
            Where = {
                School.schoolId == student.edplan.sid
             }
        }
        column={
            "课程名字"
            "课程代码"
            "课程类型"
        }
        action={
             "选课" : save
        }
        search={
             "课程编码"
             "课程名"
        }
        additional ={
             submit:checkbox
             partition:10
        }
    }
    datagrid={
        headerRow={"已选课程"}
        data={
             "课程名字" <: Schedule.course.courseName
            "课程代码" <: Schedule.course.courseId
            "课程类型" <: Schedule.course.courseType
            Where = {
                Schedule.student.studentID == user.userID
             }
        }
```

```
            column={
                “课程名字”
                “课程代码”
                “课程类型”
            }
            action={
                “退课” : delete
            }
            additional ={
                submit:checkbox
                partition:10
            }
    }
}

pageFrame basicFrame(){
    header{“JD.jpg”}
    leftBar{
        item{“选课与考试”
            #”培养计划”: eduplanPage
            #”选课选班”:classPage
        }
        item{“设置”
            #”个人信息”:personalPage
            #”修改密码”:passwordPage
            #”注销”:logoutAction
        }
    }
    footer{“footer.jpg”}
}

pageFrame loginFrame(){
    header{“JD.jpg”}
    footer{“footer.jpg”}
}
```

代码中先定义了 MainPage，参数是 loginPage。如同第二章中所述 MainPage 是工程的接入口，整个工程在运行起来后，先跳转到 loginPage，然后进行相关的跳转。loginPage 是登陆界面，定义了一个 form 标签，用户通过输入用户名和密码，点击“登陆”按钮，后台调用 loginAction，进入系统。basicPage 是用户登陆好用的显示页面，采用的是 infogrid 标签，显示学生学号，学生导师和学生学院三个信息。eduplanPage 是整个选课系统的主要页面，分为两个部分，可选课程和已选课程。可选课程供用户选课，已选课程供用户查看课程和退课使用。两个部

分都使用了 datagrid 标签。它们采用的是 datagrid 自带的 Action：save 和 delete。最后定义的是两个页面布局框架：basicFrame 和 loginFrame。其中 basicFrame 左边栏设置中调用了 logoutAction，这个是用户自己定义的 Action。

2.2.5 IDSL 语言的 Logic

IDSL 语言的 Logic 部分是用于定于业务逻辑的，它由多个 Action 组成。用户在页面点击了相应按钮、链接后，就会调用相应的 Action 来处理对应的业务逻辑。下面显示了 Action 的语法。

```
Action actionNameAction(){
    Statement
}
Statement={
    Exp|
    If EXP
    then Statement
    else Statement
    fi|
    repeat Statement|EXP until Statement|EXP|
    for (…)
    Statement
    rof|
    GotoPage()|
    GotoAction()
}
Exp={
    Set variable|
    call method
}
```

Action 是关键字，后面跟着 Action 的名字和具体定义。Action 的名字的首字母必须要大写，同时要以 Action 结尾。Action 是用来定义业务逻辑的。对于基本的业务逻辑增、删、改、查，IDSL 都可以在 View 中就能做到。但对于特殊的业务逻辑，依旧需要用户去自己定义。Action 就是给最终用户定义自己的业务逻辑用的。一般而言，普通编程语言包含顺序执行语句，条件跳转语句和循环执行语句，Action 的主体也包含着三种语句。Action 的主体定义如 statement 所示。statement 中一行代表一个语句，默认情况下语句会从上到下执行。if…then…else fi、 repeat…until、for(…) …rof 的语法和 Entity 中是一致的。在 Action 的最后，最终用户必须指定它是跳转到页面还是跳转到其他的 Action。GotoPage()表示跳转到页面，在括弧中指定页面，GotoAction 表示跳转到其他 Action，在括弧中指定其他 Action。如果最终用户在指定跳转的目的和条用语句不同，如 GotoPage

却调用了 Action，则代码生成时会报错并告知最终用户。

下面显示了选课系统中的 Action 定义。

```
Action loginAction(){
    If User.Login(user.id, user.passwd)
     then    student = Student.searchByUSERID(user.id)
          GotoPage(basicPage)
     end
}

Action logoutAction(){
     User.Logout()
     GotoPage(loginPage)
}
```

选课系统中基本的业务逻辑操作都能调用默认的 Action 来处理，需要用户自定义的 Action 就是登陆的时候调用的 LoginAction 和 logoutAction。其中，loginAction 采用了 Entity User 中定义过的 Login 方法来协助完成登陆操作。Login 方法根据用户 id 和密码来判断登陆用户是否有权利登陆系统。如果有，则会将对应的信息赋给全局变量 User。之后 loginAction 通过 IDSL Entity 中自带的 search 方法查询用户的权限，并跳转到相关页面。

2.2.6 IDSL 语言的 Restriction

Restriction 是用于在 Entity 的 Search 方法中。Restrction 的语法比较简单，下面是 Restriction 的语法：

```
RES(elem (stat*)?)
stat = AND|OR elem
elem = NOT?(Property op value)
```

elem 的语法类似 View 中 data 绑定的语法，property 是 Entity 的属性，value 是用户定义的值，op 则是相关操作，包括基本操作等于（==），不等于（!=），小于等于（<=），大于等于（>=），小于（<），大于（>）。还有 String 的操作，包括包含某一字段(contains)，以某字段开始(starts)，以某字段结尾(ends)，以及 between 表示在某范围内。Int 和 Date 类型都能使用 between。

2.2.7 IDSL 语言的预定义类型、操作和 Entity

在 IDSL 中为了方便最终用户使用，一些常用的东西都是预定义好的。

1) 基本类型

IDSL 中包括五种基本类型分别是 String、Int、Double、Bool、Date。其中 String 表示的是字符串，必须带有双引号（“”）；Int 表示整数；Double 表示小数；

Bool 表示布尔值（true 或 false）；Date 表示日期。Date 还分 Year, Month, Day, Hour, Minute, Second 六个组成部分，分别代码年、月、日、小时、分、秒。最终用户可以使用 Date 变量加“.”的操作来获取时间变量的不同部分，如 OpenDay.Year 就得到了 OpenDay 这个时间变量的年份。

2) 基本操作

基本操作有四种运算操作：+，-，*，/。对于 Int 和 Double 来说四种基本操作都能使用。+可供 String 使用，表示两个 String 类型合并起来。而 Data 只能用-操作，两个 Date 变量相减得到的也是 Date 类型的变量，表示两者之间相差的时间。Bool 操作不能使用运算操作。

基本操作还有四种关系操作：<，>，= =，!=。其中==表示等于，!=表示不等。任何基本类型都能这四种关系操作。对于 Bool 类型，false<true。而 String 类型的大小关系是按照字典顺序进行比较。

3) 容器类

容器是一种比较常用的工具。IDSL 提供了两种容器：List 和 Map。

List：List 在初始化是必须规定泛型类型。而 List 的初始化无需像 Java 那样要用 new 语句，如 List<Student> a 就初始化了一个名字是 a 的 List 容器，其泛型类型是 Student。List 有三种操作 add，remove，size。add 操作添加一个元素到 List 中；remove 删除指定的元素，如果元素不存在则不作任何操作；size 返回 List 的大小。List 取值比较方便，用标号就能取到，如对于 List<String> a, 则 a[1]就是取第一个元素。取值的小标是从 1 开始，而不是 0。

Map：Map 在初始化也必须规定泛型类型，如 Map<(String，String) > map 初始化了一个键和值均为 String 的 Map 容器。Map 有四种操作 add，remove，find，size。add 操作添加一个键值对到 Map 中；remove 删除指定键值的元素，如果此键值不存在则不作任何操作；size 返回 Map 的大小；find 返回指定键值的元素。

4) Entity Parameter

Entity Parameter 是一个特殊的 Entity，其作用是为最终用户提供控制参数，如可以根据参数名为 openTime 来控制某个页面的使用时间。Parameter 的定义如下所示：

```
Entity parameter{
    Id:String
    label:String
    content: String
    time:Date
```

```
}
```

其中 Id 是自增长号，最终用户可以不用关心。label 表示参数的名字，content 和 time 都用于表示参数的内容，但 time 是 date 类型的，而 content 是 String 类型的，最终用户可以根据需要来使用。用户可以通过 Parameter[label].content 或是 Parameter[label].time 来获取某个参数的内容。

2.2.8 IDSL 语言的变量

IDSL 的一大特点就是最终用户不用去关心变量传递的问题，一旦一个变量变申明后，最终用户可以在其它地方直接访问它。而实现这一目的的方式就是所有变量都是全局变量。一旦变量被声明后，它就会被保存在 session 中，这样最终用户就能在其它地方访问变量。但全局变量会带来两个问题：变量重名和效率低下。

1）对于变量重名，IDSL 采用的是一种树形结构来保存变量。每个变量都有自己的作用域，如一个变量 var 是申明于 basicPage 中名为 basicGrid 的 datagrid 下的 data 域，则最终用户可以通过 basicPage.basicGrid.var 来访问这个变量。每个变量分属不同的作用域，不同域中的变量不会相互冲突。但对于同一个域下的变量，如果出现重名，后出现的变量就会覆盖前面出现的变量。但 IDSL 将域划分的足够小，最终用户能较易地避免这个问题。

2）为于效率低下，IDSL 会为变量设置一个合适的 session 超时值。session 会有机制清理超时的变量，而对于那些超时的变量，IDSL 会持久化到硬盘中以免最终用户在变量超时后还会使用。所有变量均会以树形结构持久化到硬盘中。

2.3 第二层 UIDSL

通过对不同的高校信息系统进行需求调研以及实际使用，本文发现高校信息系统有一些共性，但这些共性在 IDSL 中没有很好地体现出来，或者说这些共性不能由某些语法来体现出来。比如说在每个高校信息系统中，后台数据库一定会有学生这张表，用 IDSL 表示的话它就是一个 Entity，这种特点显然不能作为某种语法来体现。所以为了体现这些共性，本文通对三个国内高校信息系统——交大研究生信息系统、交大本科信息系统和南航本科信息系统，识别出高校信息系统中的共性，并作为预定义特性。这些预定义的特性都能使用 IDSL 来定义。

由于高校信息系统业务逻辑复杂，可变性多，较难对可变性分析并作出抽象，所以本文识别和抽象高校信息系统的共性，并进行预定义，方便最终用户去使用这些共性。在 IDSL 语言的基础上，能更容易地对高校信息系统进行编程。

以下是三种预定义特性。

1) 预定义 Entity

对于高校信息系统，本文分别识别出了包括用户，角色，学生，导师，管理员，课程，班级，课表，所开课程等多种共性特性。这些预定义的特性是采用 IDSL 来实现的。一些预定义的特性如下所示：

```
Entity Course{
    courseId: String
    courseName: String
    type: String
    school:School
    tutor:Tutor
}
Entity User{
    userId:String
    passwd:String
    role:Role
    Login(id:String, passwd:String) :Bool
    Logout()
}
Entity Role{
    type:String
}
Entity Tutor{
    tutorId: String
    name: String
    school:School
}
```

当然这些预定义的特性用户是可以修改的。为了方便使用，当最终用户重写这些 Entity 时，本文会将原来 Entity 中没有的特性（属性或方法）添加到 Entity 中，而属性名一样的属性将会被覆盖，方法名一样的方法也会被覆盖。例如最终用户想对 Course 进行重写，添加一个属性——学分（score），则最终用户可以这样重写：Entity Course{ score: String }。代码生成时属性 score 将会添加到 Course 中。如果最终用户修改 Course 中的属性 type 变成 int 类型的，则只需写成 Entity Course{ type: int }这样即可。这种重载的好处在于最终用户可以只管自己所关心的属性或方法，却不需要去再写已经存在的属性或方法。而对于最终用户用不到的属性或方法，则只要不去使用即可。

2) 预定义 Page

预定义的页面本文就定义了一个 ErrorPage。其定义如下所示：

```
Page ErrorPage(message:String){
    output(message)
}
```

它基本类似于 IDSL 中 View 的定义。不同于 IDSL 中 View 的定义，ErrorPage 的定义中有 output 操作，这个表示输出消息。目前 ErrorPae 的定义完全是采用 jsp 来定义的。最终用户在调用 ErrorPage 时，只需要像 ErrorPage(“对不起，您输入的密码错误！”)这样写就可以了。最终用户也可以选择自己定义 ErrorPage，这样原先的 ErrorPage 将会被覆盖。

3) 预定义逻辑操作

预定义逻辑操作不同于 IDSL 中的 Action。由于 Action 必须指定最后跳转的页面或者 Action，这样的话耦合度过高，调用起来不是很方便。故预定义的逻辑操作是不带跳转的，它就是仅仅是处理业务逻辑用的，调用的时候只要像调用方法一样去调用这些逻辑操作即可。预定义 逻辑操作包括 autoCourse（自动排课），autoTest（自动排考试），autoScore（学积分统计）等。为了提高效率，所有的预定义逻辑操作都是用 Java 或者存储过程写的。

仍旧以选课为例子，IDSL 首先需要定义很多 Entity，Login/Logout 方法都需要用户区定义。Login 定义对最终用户用户而言是较为复杂的，因为有分支语句、Restriction 等，但如果采用的是 UIDSL 来开发这个选课系统 UIDSL 就不需要去定义很多 Entity，这些 Entity 都已经预先定义好了，连 Login/Logout 方法都是预先定义好的，这样最终用户开发起来就更加方便快捷了。若是排课，排考试系统，UIDSL 只要调用相关的 Action 就能处理好业务逻辑了。

2.4 第三层 SIDSL

2.4.1 挑战与解决方案

UIDSL 在 IDSL 的基础上添加了许多预定义的特性，使语言更加强大，同时也使得最终用户在使用的时候更方便。但对于最终用户而言，当遇上比较复杂的业务逻辑时，UIDSL 这套语言的抽象层次还比较低，粒度还比较小，用这套语言编程对最终用户而言还是有一定难度的。但鉴于无论是信息系统还是高校信息系统，其复杂性远胜于手机应用、页面标签这些有较简单 DSL 支持的领域，想在信息系统或者是高校信息系统层面上再在对语言进行抽象，使 UIDSL 更为简洁也是比较困难的。

因此，我们决定采用两种方法提高最终用户使用 UIDSL 的便捷性：1）结合可视化编程和简捷编程的技术，开发一个具有良好易用性的 UIDSL 编辑器；2）

针对子领域，设计更高抽象层次的 DSL。当领域缩小时，其可变性也随之减少，将有利于可变性的建模，让 DSL 语言支持可变性的配置或定制。

第一种方法由贾颖同学进行研究，本节将采用第二种方法，研究提出子领域 DSL 的设计方法论，并选定高校信息系统的选课子领域，对其建立 DSL——SIDSL。

2.4.2 SIDSL 设计方法

本文分了两步来建立 SIDSL：

1）分析需求确定领域的共性和可变性

对于领域的需求，通常可以通过对本领域多个系统的需求文档、遗留系统代码以及实际使用来分析与确定。通过需求分析，本文分三个方面来确定领域的共性和可变性：界面（View）、模型（Model）和逻辑（Logic）。在界面上本文采用表格的方式来展示系统的特征，在模型方面本文采用类图的方式来表示，在逻辑方面本采用 Foda 方法来描述系统的特征。

本文分析了是三个国内的高校信息系统——交大本科选课子系统、交大硕士选课子系统和南航本科选课子系统，以实际使用三套子系统以及遗留系统的文档与代码作为需求分析的基础。

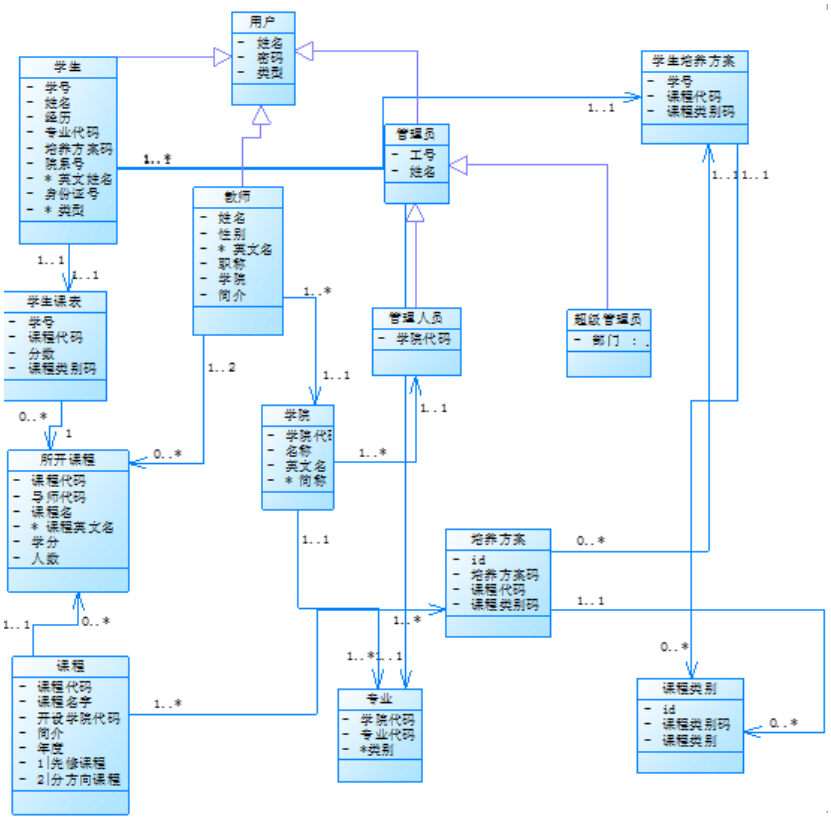

图 2-2 数据库表的领域分析

Fig. 2-2 Domain Analysis of Database Table

首先分析高校选课子系统的模型。本文采用的是类图的方式，如图 2-2，这样就能直观地表示表和表之间的关系，表中属性的类型之类的。对于类图，本文做了些改动。表中可有可无的属性用在其前面加*，若存在一组属性的情况，即一些属性必须是同时存在或没有的，在其属性前面加数字编号，并以|结束即可。如所有属性之前有数字 1 的均表示是一组属性。当然一个属性分属不同的组，可以有多个数字标号，并以逗号分隔，如 1,3,6|studentName 这样。对于可有可无的表使用虚线连接，不同的多对多的关系采用/来划分。对于数据库表，还可能存在的情况是，表和表之间的关系是每个系统都不同，如在系统一中是表 A 和表 B 有关联。，而系统二中是表 A 和表 C 有关联。这种情况下，一个类图是不好表述的，可以用多个类图来表达。本文根据实际使用情况来分析，只有一张类图。

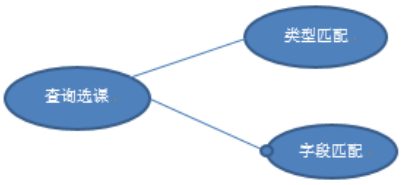

图 2-3 查询选课功能

Fig. 2-3 Search Course Function

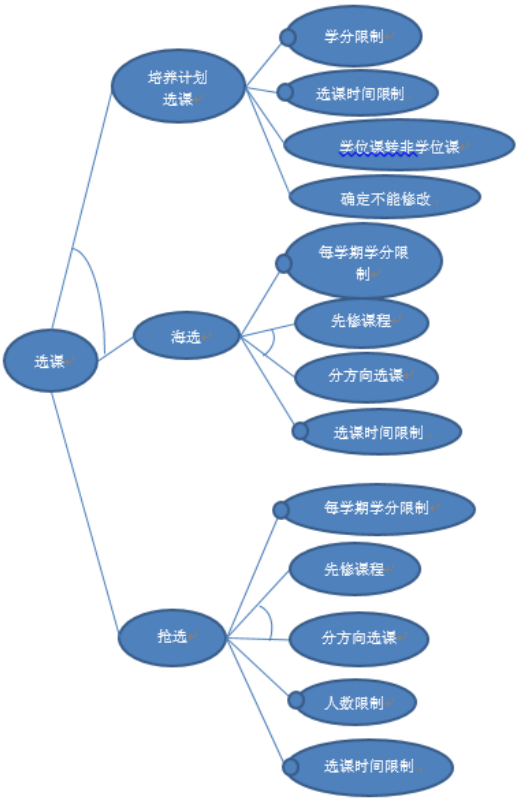

图 2-4 选课功能

Fig. 2-4 Course Selection Function

对于高校选课子系统业务逻辑，本文采用的是 FODA 的方法描述系统的特征，为了清晰起见，本文将整个系统划分为四个部分，每个部分本文都采用 FODA 图来描述，如图 2-3,2-4,2-5,2-6 所示。

FODA 中一个椭圆代表的是一种特征。两个关联的特征用线连接起来，表示一种特征是另一种特征的延生。如果某种特征是多选一的话，则是两者用弧形连起来，如图 2-4 中的培养计划和海选。若是可选的，则在特征前面加小圈，如图 2-4 中的培养计划下的学分限制。

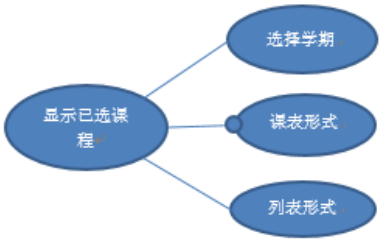

图 2-5 显示已选课程功能

Fig. 2-5 Show Selected Course Function

图 2-6 删课功能

Fig. 2-6 Delete Course Function

最后分析三个高校选课子系统选课界面的需求，使用表格的方式表示展示了高校选课子系统页面元素的共性和可变性，如表 2-1 所示。

表 2-1 页面领域分析

Table 2-1 Domain Analysis of Page

页面元素	共性	可变性
页面布局	采用拐角型布局，即上面 logo 标志栏，左侧菜单栏，右侧正文，下面辅助信息栏	logo 图片不同
		左侧菜单栏内容不同
		下面辅助信息不同
		背景色白或绿
显示字段	结构类似，都与后台数据库绑定	显示的顺序不同
		显示的字段不同
按钮	增删改查四种功能	按钮名称不同

2）统一共性并将可变性参数化以建立语言

通过第一步，领域共性和可变性被清楚地分辨出来，其共性部分由固定的架构和代码来实现，而可变性部分将转化为 SIDSL 语言，最终用户将通过 SIDSL 语言来定制这些可变性。为了使 SIDSL 更为简洁，本文采用了不同于 IDSL 的方

式来定义它。整个语言将使用 XML 模板的形式，将可变特征转化为对应的标签。不同可变性用不同方式表现出来后作为 XML 模板的属性值或者子节点。根据三种不同的图，本文也分三个部分来参数化 SIDSL 语言。

2.1）逻辑的抽象

逻辑是可以串联领域模型和界面的共性可变性的，所有要先进行抽象。逻辑中所有可选的特征或者是多选一的特征都是可变性，在语言中这些特性会以标签的形式定义在语言中，并用标签的值作为参数来体现不同的特征。同时共性特征若存在可变特征，也会以标签的形式定义在语言中，以示区分重名特征。但除了这些可变性，逻辑中每个特征还会涉及模型以及界面，这些也是会存在可变性的。需要根据不同的情况来进行参数化可变性。以图 2-3 中查询课程来说明，查询课程作为功能有两个子特征：类型匹配和字段匹配。其中字段匹配是共有特征，而类型匹配是可选特征。由于有可选特征存在，特征查询课程会作为标签<查询课程>定义在语言的 XML 模板中。可选特征类型匹配也会定义在语言的 XML 模板中。字段匹配是共有特征，但在页面显示中，它会提供查询需要匹配的字段，这些字段是不同的，故这一特征也将作为子标签进行定义。整个查询课程涉及的了模型中“课程类型”这张表，但表中没有可变性。下面讨论如何在查询课程中定义参数来区分特性的可变性。整个查询课程，可变特征有类型匹配，但最终用户需要指定类型匹配的匹配项值，这些值绑定了模型“课程类型”中字段“类型”的值，同时有多个匹配项的情况也是存在的，所以在<类型匹配>这个标签下添加了<项>这个子标签，同时标签<项>中的值以 String:String 的形式来绑定模型中的值。

2.2）模型的抽象

模型的抽象针对的是未在逻辑抽象中参数化的模型可变特征。选课子领域模型的可变性体现在表字段的可选或多选上。这些字段同样会以标签的形式定义在 IDSL 语言中，字段所在的表也要以标签的形式定义在 IDSL 语言中，同时字段标签会以子标签的形式定义在表标签之下。对于可选特征，字段标签的值若为 1，则表示体现该特性。对于多选特征，则字段标签的值表示选择对应的特性。如 1 选择第一种特性，2 选择第二种特性。所有这些标签都要定义在<实体>标签之下。

2.3）页面特征的抽象

经过逻辑的抽象，与逻辑相关的页面元素的可变性都会被参数化，除了而页面布局。这部分的可变特性相互之间没有联系，只要将可变特性定义为标签即可。

以上述方法定义出一个 XML 模板，所有标签值都默认为 0。用户在使用时只要在对应的标签中填写自己想要的值即可。但所有标签一起填写是相当麻烦的，故每个标签都有对应一个编号，最终用户定义语言时只要指明编号，再指明对应

的值即可。形如 1：“eduplan”就是指明了标签页面名字的值是“eduplan”。

2.4.3 SIDSL 语言的定义

根据上一节，SIDSL 的 XML 模板如下所示：

```
<页面名字>String</页面名字><!--- 编号 1 --->
<查询课程><!--- 编号 2--->
    <字段匹配><!--- 编号 2.1--->
        <项>String:Property</项>
    </字段匹配>
    <类型匹配><!--- 编号 2.2--->
        <项>String:String</项>
    </类型匹配>
    <结果显示><!--- 编号 2.3--->
        <项>String:Property</项>
    </结果显示>
</查询课程>
<选课><!--- 编号 3 --->
    <培养计划选课> <!--- 编号 3.1 --->
        <分数限制>Int</分数限制> <!--- 编号 3.1.1 --->
        <时间限制>Date-Date</时间限制><!--- 编号 3.1.2 --->
        <按钮>String</按钮><!--- 编号 3.1.3 --->
    </培养计划选课>
    <海选><!--- 编号 3.2 --->
        <分数限制>Int</分数限制> <!--- 编号 3.2.1 --->
        <时间限制>Date-Date</时间限制><!--- 编号 3.2.2 --->
        <先修课程>0|1</先修课程><!--- 编号 3.2.3 --->
        <分方向”>0|1</分方向><!--- 编号 3.2.4 --->
        <按钮>String</按钮><!--- 编号 3.2.5 --->
    </海选>
    <抢选><!--- 编号 3.3 --->
        <分数限制>Int</分数限制><!--- 编号 3.3.1 --->
        <时间限制>Date-Date</时间限制><!--- 编号 3.3.2 --->
        <先修课程>1</先修课程><!--- 编号 3.3.3 --->
        <分方向”>1</分方向><!--- 编号 3.3.4 --->
        <人数限制>Int<人数限制><!--- 编号 3.3.5 --->
        <按钮>String</按钮><!--- 编号 3.3.6 --->
    </抢选>
</选课>
<显示已选课程> <!--- 编号 4 --->
    <课表显示>0|1</课表显示><!--- 编号 4.1 --->
    <列表显示><!--- 编号 4.2 --->
        <项>String:Property</项>
    </列表显示>
```

```
</显示已选课程>
<删课> <!--- 编号 5 --->
    <按钮>String</按钮><!--- 编号 5.1 --->
</删课>

<实体><!--- 编号 6--->
    <学生><!--- 编号 6.1--->
        <英文名>0|1</英文名><!--- 编号 6.1.1--->
        <类型>0|1</类型><!--- 编号 6.1.2--->
    </学生>
    <教师><!--- 编号 6.2--->
        <英文名>0|1</英文名><!--- 编号 6.2.1--->
    </教师>
    <学院><!--- 编号 6.3--->
        <简称>0|1</简称><!--- 编号 6.3.1--->
    </学院>
    <专业><!--- 编号 6.4--->
        <类别>0|1</类别><!--- 编号 6.4.1--->
    </专业>
</实体>

<页面布局><!--- 编号 7 --->
    <logo 图片>String</logo 图片><!--- 编号 7.1 --->
    <左边栏><!--- 编号 7.2--->
        <项> #*String：URL </项>
    </左边栏>
    <背景色>1|2</背景色><!--- 编号 7.3 --->
    <页脚>String</页脚><!--- 编号 7.4 --->
</页面布局>
```

因为是从中文系统中抽象出来的，故所有的标签名都定义为中文，XML 也支持中文。标签的分层体现了特性的层次结构。第一个标签是页面名称，这个定义了生成页面的名字。标签中的值如果是 0|1，则是二选一。如果是 String，Int，Date 则是 SIDSL 的基本类型。String 表示的是字符串，必须带有双引号（“”）；Int 表示整数； Date 表示日期，形式是 YY:MM:DD:HH:MI:SS，表示年、月、日、小时、分、秒。在标签<查询课程><字段匹配><项>中的值 String：String 中，前面的 String 表示的是在页面上显示的字段名字，后面的 String 表示模型中表“课程类别”的“课程类别码”字段所对应的值。在标签<查询课程><字段匹配><项>的值 String：Property 中，前面的 String 表示的是在页面上显示的字段名字，后面的 Property 表示模型中表“课程”字段名。<logo 图片>和<页脚>中 String 值表示 pic 的地址。在标签<左边栏><项>中的值 String：URL 中，String 表示的是在页

面上显示的字段名字，URL 表示跳转。没有“#”是根链接，有一个“#”则是一级链接，以此类推。子连接会在开始显示的时候收缩在父链接下。<背景色>1|2</背景色>中 1 表示白，2 表示绿色。所有的形如 0|1 这样的选择，都默认选择第一个值。其它若是可选特征的标签，用户没有定义，则默认没有选择该特征。

下面显示了选课系统的 SIDSL 代码示例。

```
1:”eduplan”
7.1: “JD.jpg”
7.2:<项>“选课与考试”</项>
    <项>#”培养计划”: eduplanPage</项>
    <项>#”选课选班”:classPage</项>
    <项> “设置”</项>
    <项>#”个人信息”:personalPage</项>
    <项>#”修改密码”:passwordPage</项>
    <项>#”注销”:logoutAction</项>
7.4:“footer.jpg”
2.1:<项>““课程名字” : 课程.课程名字</项>
    <项>““课程代码” : 课程.课程代码</项>
2.3：<项>““课程名字” : 课程.课程名字</项>
    <项>““课程代码” : 课程.课程代码</项>
    <项>““课程类型” : 培养计划.课程类别码</项>
3.1.3:”选课”
4.2:<项>““课程名字” : 课程.课程名字</项>
    <项>““课程代码” : 课程.课程代码</项>
    <项>““课程类型” : 学生课表.课程类别码</项>
5.1:”删课”
```

对于 SIDSL，最终用户无需去了解 Model、View 和 Logic，最终用户只要知道 XML 模板中每个标签的含义，定义程序时需要使用什么标签，程序中对应标签的值就能编写相关的程序，这比 IDSL、UIDSL 都方便。

2.5 本章小结

本章首先分析了高校信息系统的开发维护中遇到的问题，阐明了对高校信息系统建立最终用户的领域特定语言的优势和可行性。然后根据信息系统的特点，提出了采用分层的方法来定义信息系统的 DSL 语言的思路。将语言分为三层：信息系统层、高校信息系统层和选课子系统层。这三层语言的作用领域是逐渐缩小，但语言的抽象层次是不断提升的。对于这三层不同的语言，本章在描述过程中都是先给出了定义它们的方法论来，之后再给出了它们各自的用法。本章也以选课系统的例子来进一步说明三层语言的不同用法，同时还作了比较，说明三层

语言是不断简洁，越来越方便最终用户编程的。

第三章 领域特定语言到 Java 的代码生成工具的开发

3.1 领域特定语言到 Java 的代码转换的总体技术方案

3.1.1 技术路线

让一种编程语言运行方法一般有两种，一种是开发该语言的解释器，对源代码进行解释执行。如 basic，它们是边扫描边翻译，逐句输入逐句翻译，逐行进行执行，并不产生执行程序。另一种是开发该语言的编译器，根据代码生成相应的 EXE 文件。如 PASCAL、FORTRAN，它们的编译器会将整个源程序转化为用机器码写的等价于源程序的目标程序，然后执行目标程序。本文采用第三种方法，将领域特定语言程序转换成 Java 代码，再依赖 Java 虚拟机进行执行。图 3-1 显示了三层语言的代码转化过程。

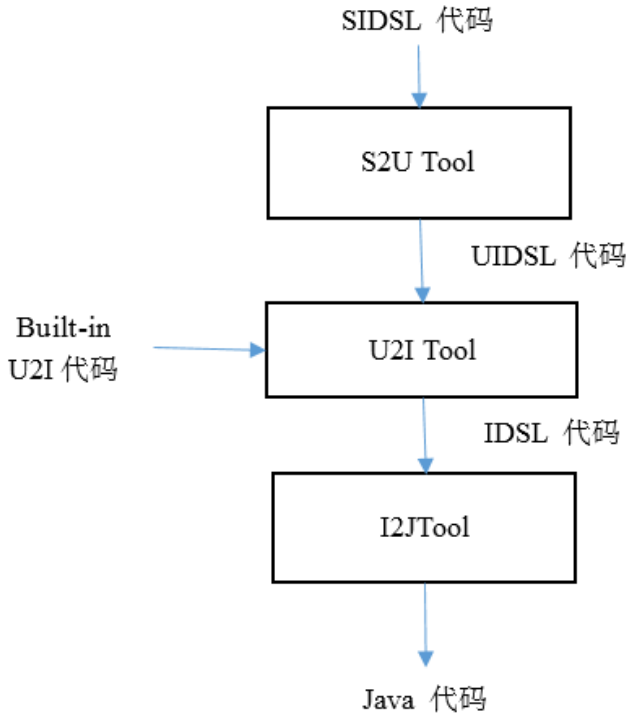

图 3-1 代码转化框架

Fig. 3-1 Code Generation Framework

S2U 工具把 SIDSL 代码转化为 UIDSL 代码。S2U 首先会读取最终用户文件并逐行分析，将值填写到对应的 XML 模板中，形成一个完成的 SIDSL 代码。然后根据 XML 的不同值将 SIDSL 代码转为 UIDSL 代码。U2I 工具把 UIDSL 代码转化为 IDSL 代码。I2J 工具是把 IDSL 代码转化为 Java 代码的工具。

以上三个工具均采用 Java 开发，并基于 Eclipse 插件——Spoofax 实现了到 Java 代码的转换。其中，I2J 工具是基础，也是核心，本章以下各节将重点针对这一工具进行详细设计与实现。

3.1.2 基于重写规则的程序转换

程序转换技术目前有很多研究成果，比较成熟的程序转换技术有以下三种[1]：

1) 基于转换器的代码生成

基于转换器的代码生成，是通过读取“语义模型”作为输入，然后为目标环境生成源代码。转换器通常分为两个部分：输入驱动和输出驱动。输出驱动的转换从输出结果入手，然后按需寻找对应的输入数据；而输入驱动的转换则整体读取输入数据结构，然后再生成输出。

当输出文本和输入模型有简单的映射关系且大部分输出文本可以自动产生时，基于转换器的代码生成很容易编写而且不需要引入任何模板工具。

2) 模板化生成器

模板化的生成器的基本思想是编写希望的输出文件，然后插入标注（callout）以表示可变部分。然后使用具备模板文件模板处理器和相应语境，通过上下文填写标注进而生成真正的输出文件。

模板化的生成器最大的好处在于，通过阅读模板文件，就可以很容易了解最终输出的样子。因此，模板化的生成器特别适用于当输出包含大量的静态内容，而仅仅有少量简单的动态内容的时候。

3) 基于重写规则的程序转换

重写规则是一种范式描述，通过对范式左部进行模式匹配，并用右部替换匹配成功的部分。一个基本的重写规则有如下形式：

R: p1->p2

这里 R 是规则的名称，这个规则含义是当遇到模式 p1 时，将使用模式 p2 进行替换，模式是可以携带变量的，如：

PlusZero : Plus(e,Int(0)) -> e

这个规则的含义是 e+0 的结果，这里的 e 就是变量。

本文研究提出的 IDSL 语言，所面向的是高校信息管理系统领域，输入模型复杂，逻辑可变性高，通过分析比较以上三种程序转换技术可知，基于重写规则的程序转换方法是通过对抽象语法树的分解，再根据分解的粒度即子树的大小控制可变部分的表达能力，这非常适合复杂的程序转换系统的实现。因此，本文选

取了基于重写规则的转换技术，研究和开发从 IDSL 程序到 Java 代码的转换工具。

基于重写规则的程序转换流程如图 3-2 所示，第一步，在管道的源端，读入输入的程序文本，并将其转换为解析树或抽象语法树；第二步，将语法树作修改变换，达到目标语言的抽象语法树；最后，将输出树再转换为程序文本。

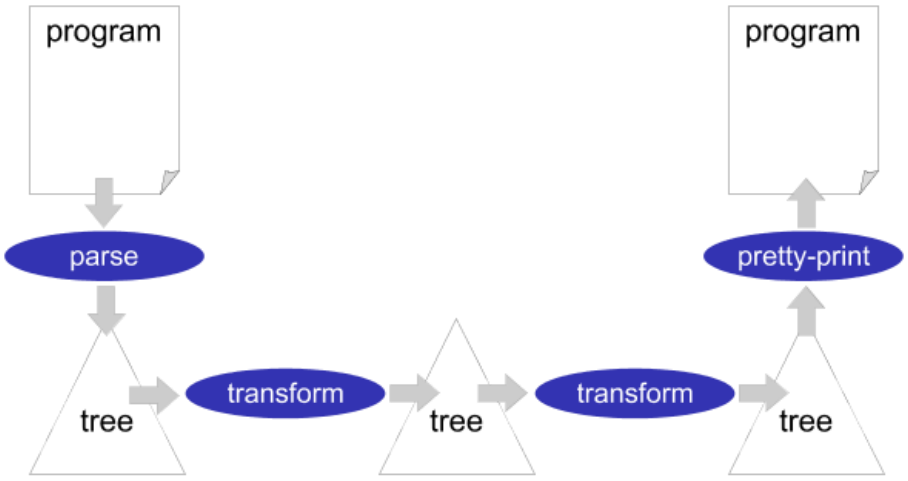

图 3-2 基于重写规则的程序转换流程

Fig. 3-2 Program Conversion Process Based on Rewrite Rules

目前常见的基于重写规则的程序转换框架有 Stratego/XT[26][28]和 XText。本文将选取 Stratego/XT 框架作为程序转换的核心框架，Stratego/XT 的优势在于，除了支持传统的重写规则外，还具备独特的重写策略支持，可以实现复杂的程序转换系统；另外，Stratego/XT 内置了丰富的规则来完成字符串操作、文件读写等常见操作，并且内置有语法解析器、打印器等工具，使用非常方便。

Stratego/XT 框架包括两个部分，一是用来描述转换规则的语言 Stratego，另一个则是一个工具集合 XT[30]，包括解析器(Parser)和打印器(Pretty-printer)。Stratego/XT 用于以一种高层次的抽象提高转换系统的效率，通过 Stratego/XT，本文可以构造出编译器、文档生成器等应用[26]。

在基于 Stratego/XT 的转换系统中，所有的转换规则都必须以 Stratego 语言来描述。Stategо 提供了基本的重写规则(basic rewrite rule)来满足基本的转换，还提供了可编程的重写策略(programming rewrite strategy)来控制规则的执行顺序。同时，为了描述上下文相关的转换，Stratego 也提供了动态重写规则(dynamic rewrite rule)[27]。

3.2 IDSL 到 Java 的代码生成工具的总体设计

3.2.1 代码生成工具的架构

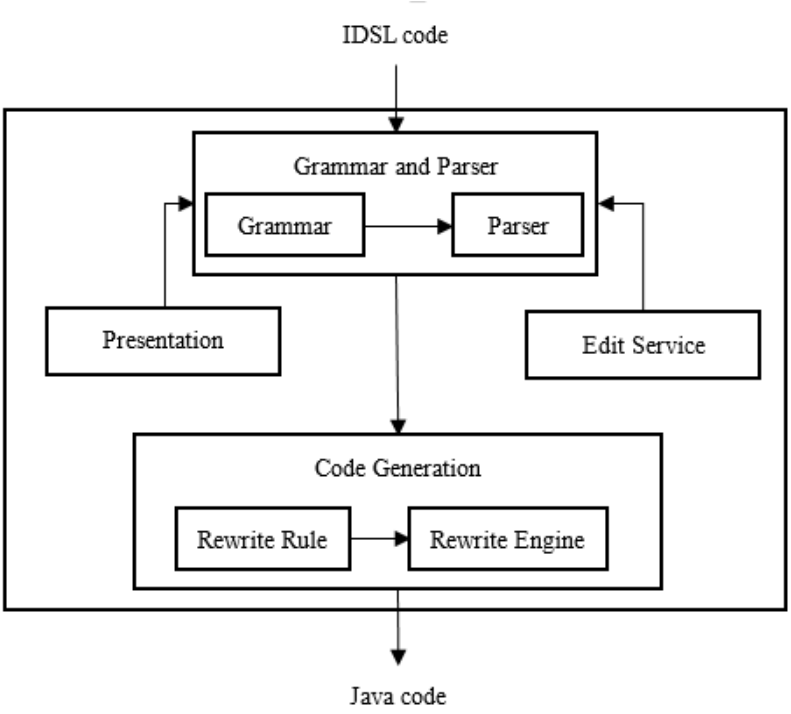

图 3-3 I2J 工具架构
Fig. 3-3 I2J Tool Architecture

图 3-3 是基于 Spoofax 平台的 I2J 工具架构。其中 Grammar and Parser 模块是整个平台的基础，它会将用户自定义的语法解析成对应的 parse table，为其他三个模块提供了语法依据； Code Generation 模块中使用 Stratego 编写的重写规则，经过编译后，实现 Java 代码的生成。Presentation 模块会根据 parse table，实现代码高亮，代码折叠等功能；Edit Service 模块提供了编辑过程中的语法结构补全、括号匹配、添加注释等功能。

3.2.2 基于 Eclipse 的 Spoofax 插件

本文采用了基于 Eclipse 插件的体系结构来实现 I2J 工具，并选择了开源项目 Spoofax 作为平台。

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台，其本身是一个精心设计的框架和一组服务，用于通过插件构建开发环境[31]，如图 3-3 所示。其中 Workbench 组件包含了一些扩展点，例如，允许插件扩展 Eclipse 用户界面，使这些用户界面带有菜单选择和工具栏按钮；请求不同类型事件的通知；以及创建新视图。Workspace 组件包含了与资源（包括项目和文件）交互的扩展点。

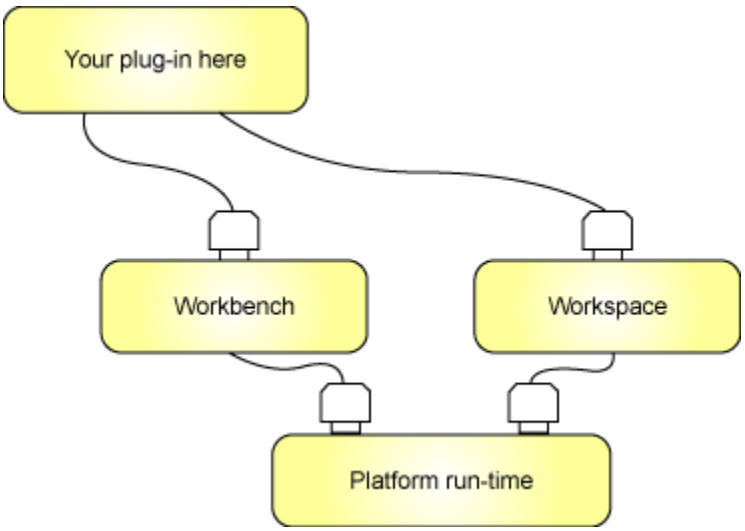

图 3-4 Eclipse Workbench 和 Workspace：必备的插件支持
Fig. 3-4 Eclipse Workbench and Workspace: basic plugin support

本文采用开源项目 Spoofax[32]作为 I2J 工具的实现平台，Spoofax 是一个基于 StargoXT 的 Eclipse 的插件，提供了一套语言结构，用于定制转换工具的外观、语法、转换规则等。

3.2.3 Spoofax 平台下 IDSL 语言结构

表 3-1 基于 Spoofax 平台的 IDSL 语言结构
Table 3-1 IDSL Structure Based on Spoofax

Custom	Generated
Syntax definition	
IDSL.sdf	Common.sdf
Editor service descriptors	
IDSL.main.esv	
IDSL-Builders.esv	IDSL-Builers.generated.esv
IDSL-Colorer.esv	IDSL-Colorer.generated.esv
IDSL-Completions.esv	IDSL-Completions.generated.esv
IDSL-Folding.esv	IDSL-Folding.generated.esv
IDSL-Outliner.esv	IDSL-Outliner.generated.esv
IDSL-References.esv	IDSL-References.generated.esv
IDSL-Syntax.esv	IDSL-Syntax.generated.esv
Semantic definition	
idsl.str	
check.str	
generate.str	

表 3-1 将 Spoofax 下的文件进行了分类，其中 Syntax definition 部分是 IDSL 语法的 SDF 定义。Editor service descriptors 部分则是 Spoofax 内定的编辑器描述

语言，这些语言根据规则去描述编辑器的外观、编辑服务等。Semantic definition 部分是使用 Stratego 语言的转换模块，用于检测重写规则。Spoofax 的一个重要的设计原则是将生成的文件与用户手写的文件结合，并通过文件名中的“generated”进行区分，每一个生成的文件都会在工程编译时被重新生成。

图 3-4 显示了 Spoofax 工程下的目录结构。其中目录 Syntax 下是用户自定义 SDF 文件。 目录 trans 下是用户自定义的重写规则文件。目录 editor 下则是描述编辑器外观、服务文件，这些文件是按照默认配置生成的，但用户可以根据需求进行修改。

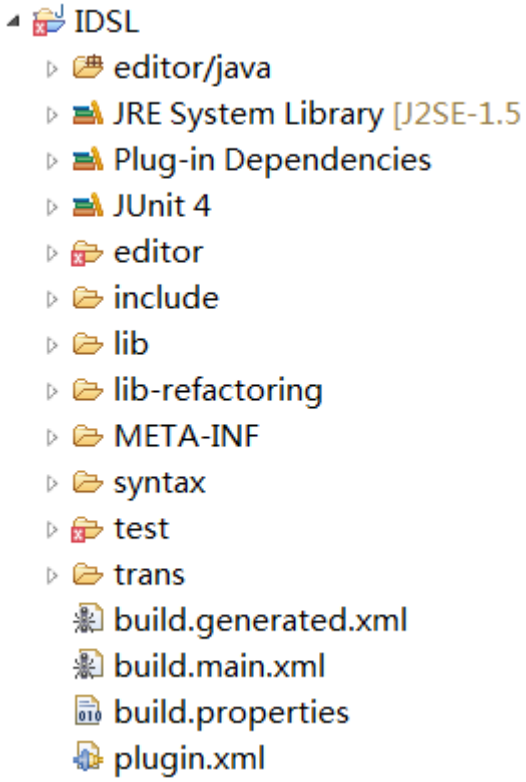

图 3-5 Spoofax 工程的目录结构

Fig. 3-5 Directory Structure in Spoofax Project

3.2.4 代码生成工具的工作流程

图 3-6 是 I2J 工具的界面截图。用户只需定义好自己的 ids 文件，点击 Transform 中对应的按钮就能转化为最终的代码。其中 IDSL 文件的关键字高亮由 Spoofax 的 Presentation 模块提供，语法提示由 Edit Service 模块提供。

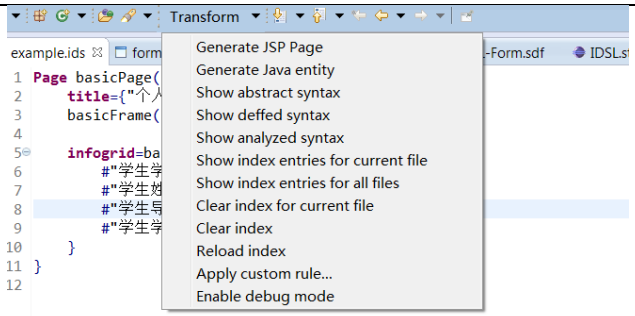

图 3-6 I2J 工具的界面截图

Fig. 3-6 Screenshots of I2J Tool

基于 Spoofax，I2J 工具的程序转换主要分为两步：

1.语法规则编译

1.1 从 IDSL 语法定义文件 syntax/idsl.sdf 中，生成解析表 include/idsl.tbl；

1.2 从 IDSL 规则文件 trans/idsl.str 中，生成编译后的规则 include/idsl.ctree 。

2. 转换规则执行

2.1 解析用户编写的 IDSL 文件，根据定义的语法，给出错误信息；

2.2 对 IDSL 文件进行静态分析，调用 editor/IDSL-Builders.esv 中 observer 指定的规则进行分析(如图 3-7)，如 observer:editor-analyze。因此，editor-analyze 规则需在 trans/idsl.str 中实现，该规则用于返回分析后的抽象语法树(AST)、警告信息、错误信息等。

```
imports IDSL-Builders.generated

builders

  // This file can be used for custom analysis and builder rules.
  //
  // See the imported file for a brief introduction and examples.

builders

  provider : include/idsl.ctree
  provider : include/idsl-java.jar

  observer : editor-analyze                          (multifile)

  builder  : "Generate JSP Page"                     = generate-jsp (openeditor) (realtime) (meta) (source)
  builder  : "Generate Java entity"                  = generate-entity (openeditor) (realtime) (meta) (sour
  builder  : "Show abstract syntax"                  = debug-generate-aterm (openeditor) (realtime) (meta)
  builder  : "Show deffed syntax"                    = debug-generate-deffed (openeditor) (realtime) (meta)
  builder  : "Show analyzed syntax"                  = debug-generate-analyzed (openeditor) (realtime) (met
  builder  : "Show index entries for current file" = debug-index-show-current-file (openeditor) (realtime
  builder  : "Show index entries for all files"      = debug-index-show-all-files (openeditor) (realtime) (
  builder  : "Clear index for current file"          = debug-index-clear-current (meta) (source)
  builder  : "Clear index"                           = debug-index-clear (meta) (source)
  builder  : "Reload index"                          = debug-index-reload (meta) (source)

  on save  : editor-save
```

图 3-7 IDSL-Builders.esv 文件结构

Fig. 3-7 DSL-Builders.esv File Structure

2.3 当在 trans/idsl.str 中编写了转换规则 generate-java 后，通过在 editor/IDSL-Builders.esv 中指定规则：

Builers:"Generate Java Code" = generate-java

即在 Transform 菜单下出现"Generate Java Code"的菜单项，如图 3-6，点击菜单项，Spoofax 会调用相应的规则来处理静态分析后的抽象语法树。此外，也可以将规则定义在 editor/IDSL-Builders.esv 中 on-save 项下：

on-save : generate-java

这样 Spoofax 会在用户保存文件时自动调用规则。

3.3 IDSL 到 Java 的代码生成工具的转化规则设计

本节首先会根据 IDSL 的语法，给出其相关 SDF 的描述，然后通过对 Java EE 工程中常用的编程模式进行抽象，找出从 IDSL 语句到 Java EE 代码的最为简洁、方便的映射关系，再通过 Stratego 语言和其提供的相关 API 对映射关系进行描述，写出对应的重写关系，从而设计出 I2J 工具的转换规则。

3.3.1 IDSL 的 SDF 描述

在设计转化规则前，首先要定义出 SDF 这个文件。具体 SDF 的语法比较简单，如图 3-8 所示。一行表示一个 SDF 语法，每行箭头的左侧是相关语法定义的形式，右侧则是对语法定义的形式进行取名。其中红字 cons 内部的名字是 Spoofax 在生成抽象语法树时所用的名字，供系统所用，这个名字是不能重复的。而箭头右侧第一个名字则是供用户使用，即用户在定义 SDF 文件时，遇到类似的语法形式，则可以使用这个名字来代替，它是可以重复的。

```
"Page" PageId "()" "{" PageTitle FrameCall PageElement*"}" -> Page{cons("PageTypeA")}
"Page" PageId "()" "{" PageTitle PageElement*"}" -> Page{cons("PageTypeB")}
"title" "=" Id "{" STRING "}" ->PageTitle{cons("PageTitleTypeA")}
"title" "=" "{" STRING "}" ->PageTitle{cons("PageTitleTypeB")}
```

图 3-8 sdf

Fig. 3-8 sdf

Spoofax 会自动将所有 Syntax 目录下的文件进行整合，然后生成 idsl.str 这个文件，如图 3-9 所示。这个文件根据用户自己定义的 SDF 将其转化为抽象语法树，并以文件的方式展示出来。文件中一行就代表了抽象语法树中的一棵子树。每行冒号左侧就表示子树的名字，而右侧是一个表达式。很容易看出来，表达式箭头的左侧是这棵子树的子节点，箭头的右侧则是 SDF 文件中供用户使用的语法定义的名字。这是为了方便用户使用这个文件而这样设计的。

```
constructors
                                : Sort -> Sort1
                                : Infogrid -> PageElement
                                : Form -> PageElement
                                : Datagrid -> PageElement
  FrameCall                     : FrameId -> FrameCall
  PageTitleTypeB                : STRING -> PageTitle
  PageTitleTypeA                : Id * STRING -> PageTitle
  PageTypeB                     : PageId * PageTitle * List(PageElement) -> Page
  PageTypeA                     : PageId * PageTitle * FrameCall * List(PageElement) -> Page
  InfogridData                  : List(SingleInfogridData) -> InfogridData
  SingleInfogridData            : STRING * DataOperation * EntityProperty -> SingleInfogridData
  Photo                         : STRING -> Photo
  InfogridTypeD                 : List(InfogridData) -> Infogrid
  InfogridTypeC                 : Photo * List(InfogridData) -> Infogrid
  InfogridTypeB                 : Id * List(InfogridData) -> Infogrid
  InfogridTypeA                 : Id * Photo * List(InfogridData) -> Infogrid
  FormB                         : DataSchema * ColumnSchema * ActionSchema -> Form
  FormA                         : Id * DataSchema * ColumnSchema * ActionSchema -> Form
  submitTypeB                   : AddtionalType
  submitTypeA                   : AddtionalType
  settingTypeB                  : AddtionalType
  settingTypeA                  : AddtionalType
  partition                     : Int -> AddtionalType
  order                         : STRING -> AddtionalType
  ok                            : ActionId -> ActionType
```

图 3-9 idsl.str 文件

Fig. 3-9 idsl.str File

3.3.2 基于 SSH 框架的 Java EE 工程抽象

1) Entity 的抽象

Entity 在 IDSL 中对应了 SSH 中的数据库操作，包括 Java bean、hibernateDAO、hibernateService 等一系列操作。

1.1）持久化实体类要素的抽象

Entity 除了方法（function）以外，其他部分在 SSH 中就对应了 Java bean 类。所有 Java bean 类中除了属性以外还需要有 setter/getter 方法，同时为了和数据库关联还需要由 XML 对 bean 中的属性和数据库表中的字段进行一一绑定。这种方式相对繁琐而且代码转化的量较大，这边本文采用的是注解的方式生成 Java bean 类。JPA 是 ORM 上标准的注解 API，通过结合 Hibernate 的注解方式，繁琐的 XML 配置可以免去，简化了代码的生成。以下便是抽象出使用这种配置方式的一些基本元素。

Entity：每个用于持久化的 Entity 类都需要使用@Entity 注解，使得 ORM 的框架能够将其持久化到数据库中：

```
@Entity
public class User{
public User(){}
//properties
}
```

Identity：每个 Entity 都必须有一个唯一的值作为其主键，其值是自增长的，不需要用户去关心的。本文使用@Id 来标记主键变量， @GeneratedValue 来定义主键的自增长方式：

```
@Id@GeneratedValue
private Long id;
public Long getId(){return id;}
public void setId(Long id){this.id = id;}
```

Property：Entity 类内的每一个属性都应具备 setter/getter 方法，并与数据库中的字段应该是一一对应的 ：

```
private String username;
public String getUsername(){return username;}
public void setUsername(String username){this.username = username;}
```

Associations：当属性的类型是其他类型的 Entity 或 Entity 集合时，便存在了 Entity 与 Entity 之间的关联关系：

```
@ManyToOne
private Role role;
public Role getRole(){return role;}
public void setRole(Role role){this.role = role;}
```

1.2）HibernateDAO 的抽象

根据 SSH 的分层，Struts 中 Action 类的方法应该调用 Service 类方法，Service 类再调用 DAO 类方法，不建议跨层调用。一般的，比较基本的方法会被封装成 DAO 类的方法，这些方法应该是直接操纵数据库的方法，比如增、删、改、查这些方法。而比较复杂、跨多表的方法应该封装成 Service 类的方法，这些方法是有 Action 类的方法调用，一般做些复杂的数据库操作，需要 DAO 类中的多个方法一起使用。一般而言，用户在 Entity 中定义的方法，用户总会在 Action 中使用，属于 Action 类的方法直接调用，这些方法本文就定义在 Service 类中。而其它的方法，如增、删、改、查的方法统一放到 DAO 类中。当然这些方法在 Service 类中也会存在，但这些方法就直接调用类 DAO 中的相应方法。虽然在 IDSL 代码转化为 Java 时，定义两层——DAO 层和 Service 层增加了代码转化的量和复杂性，但这样做也是有好处的。这样的分层比较规范，降低一定的耦合度，专业的程序员在修改，调用代码的时候能比较方便。当然，为了降低了转化时的代码

量和复杂性，本文让所有的 DAO 类继承自父类——HibernateDao[33]。这样公用的方法如增、删、改、查都能在父类中实现，子类只需要简单的继承下 HibernateDao 即可。表 3-2 是 HibernateDao 类的一些操作和相关说明。HibernateDao 还采用了泛型参数，并封装了常用的方法，供 Service 层调用。

表 3-2 HibernateDao 类中封装的操作及其说明

Table 3-2 Explanation for Build-in Operation in Hibernate

方法名	说明
public <X> X get(final Class<X> clazz, final Serializable id)	根据实体类与 ID 获得对象
public T get(Serializable id)	根据 ID 获得对象
public void delete(final Object entity)	删除对象
public void delete(final Serializable id)	根据 ID 删除对象
public void delete(final Class clazz, final Serializable id)	根据实体类和 ID 删除对象
public void save(final Object entity)	保存对象
public List<T> getAll()	获取所有数据
public List<T> query(final Criterion... criterions)	根据条件获取数据
public Query createQuery(final String hql, final Object... values)	HQL 方式查询
public SQLQuery createSQLQuery(final String sql,final Object... values)	SQL 方式查询
public Criteria createCriteria(final Criterion... criterions)	对象化查询

2）View 的抽象

2.1）Form 表单

View 中最主要的部分在于 Datagrid，Form 和 Infogrid 三种表现形式的转化。由于他们都是包含 Action 操作的，在代码生成的过程中，这些 Action 都是需要转化为 Struts 中的 Action 类的。在 SSH 框架中，由页面跳转到执行后台 Action 类的方法之一就是提交 Form 表单，同时页面中的参数也随 Form 表单一起提交给后台 Action 类。所以 Datagrid，Form 和 Infogrid 三者的生成的代码是被 Form 表单封装起来，而 action 标签域的相关操作会响应这个 Form 表单。其中每个 Form 表单都需要和后台 Action 类的一个方法绑定，以确定提交此表单后具体业务逻辑操作。这里采用“类名!方法名”的方式绑定 Form 表单所对应的 Action 类中的方法，这种方式的好处在于不需要在 Struts 的配置文件中绑定方法，降低了重写规则的复杂度。

除了页面上要封装在 Form 表单中的内容外，View 中的默认 Action 操作 save，update，delete，search 都需转化为后台 Action 类。因为只有默认的四种操作，故

所有生成的 Action 都统一继承自父类——StrutsAction，将增、删、改、查以及和页面交互传递参数的方法都在这个父类中定义出来，这样 Action 类只需要简单地继承这个父类就行了。同时 StrutsAction 还采用了泛型参数,，进一步降低了代码的耦合度。

2.2）Struts 标签

除了传统 HTML 标签，SSH 工程的页面还将采用 Struts 标签，这是动态页面标签，如 datagrid 中 data 域筛选出来的结果基本上都是多条记录，是个 List。而 Struts 标签可以通过 s:iterator 来对整个 List 进行遍历来显示每条记录，这样无论 List 内容的多少都能使用相同的页面代码来显示。使用 Struts 标签好处除了能显示动态内容以外，它和 Struts 的 Action 类也是有关联的，这样就统一了 Action 传递参数的方式，简化了重写规则的复杂度。

3）Action

这里探讨的 Action 类是用户自定义的 Action 类，没有默认的方法。故本文不会让这些 Action 类继承父类，它们只需要实现 Preparable 这一接口即可。Prepare 作为 Struts 中有一个重要的拦截器，当 Action 实现了 Preparable 接口，这个拦截器就会在 execute()方法执行之前调用 prepare()方法完成前置逻辑处理。Prepare 拦截器还有另一种形式 prepareMethodName，即在 prepare 后加上需要拦截的方法名。如当需要拦截 input 方法时，可以写成 prepareInput。这样，通过在执行 input 方法前，会先执行 prepareInput 中的逻辑。

4）配置文件

SSH 工程中配置文件是比较重要的，它们将不同部分结合起来，同时也能有效的降低耦合度。一般而言，SSH 工程主要配置四个配置文件：Hibernate 的 Hibernate.cfg、Struts 的 Struts.XML、Spring 的 ApplicationContext.XML 和 Web.XML。其中 Web.XML 的配置基本固定，可以预先确定好。Hibernate 的配置文件已经使用 Entity 注解的方式替代了。对于 Struts 的配置文件，本文采用默认配置——struts-default。让这整个工程都使用这个默认配置，包括页面拦截器等操作。这样的好处在于 Struts 的配置文件能都在工程一开始就确定下来。对于 Spring 的配置，本文采用的是 AOP 注入的方式配置文件。这样不仅能有效地减少耦合度，也能避免对配置文件的修改方便代码重写。

3.3.3 使用 Stratego 语言描述转换规则

根据上一节中对 Java EE 编程模式的抽象，本节将研究使用 Stratego 语言描述转换规则。一个 Stratego 语言描述的规则格式如下：

```
rule-name:
    left(param) ->$[
    result/[param']
]
with
    param':= <another-rule>param
```

rule-name 是规则名，这个规则名在 Stargo 的用户手册中没有指明命名规范，但遵从其例子中的命名，以字母和横杠“-”来命名。left(param)是匹配的项，相当于抽象语法树中的子树。其中 left 是匹配项的名字，param 是匹配项的参数，即子树的子节点。只有在 idsl.str 这个文件中出现过的子树才能做为匹配项，具体细节可见 3.4.1。$[]中则是转换后得到的内容，result/[param']是转换内容的表现形式。其中 result 表示直接的转换内容，它们在最后显示时是直接显示的，而[param']表示的是变量，它们在最后显示时会被替换掉，这些变量可以是匹配项中自带的变量 param，也可以是 with 中声明的变量。直接显示的内容与变量区别就在于[]，它就类似于 Java 中\这个转义符号，而且这个符号无法通过转移符号表示出来，故[]无法显示出来。with 部分则是对[param']的声明。如果存在多个变量时，可以有多条变量声明语句，语句间以分号来区分。但最后一条声明语句不能有分号。

1) Entity 的重写规则

根据 SSH 框架的结构，转换过程中需将 Entity 的属性和方法分离，将属性分离到用于持久化的 Java Bean 类中并添加相关的注解，将操作分离到单独的 DAO 中或者是 Service 类中，具体的转换规则如表 3-3 所示：

表 3-3 IDSL Entity 抽象语法树到 Java 代码的转换规则

Table 3-3 Rewrite Rules from IDSL Entity Abstract Tree to Java Code

IDSL 的抽象语法树	Java 语句

Entity(x,body)	public class [x]{
	[prop]
	}

Entity(x,body)	public class [x]DAO extends HibernateDAO{
	[func]
	}

Entity(x,body)	public class [x]Manager{
	[func]
	}

表 3-3（续）

PropertyNoAnno(x,propkind,srt)	private [srt] [x];
	public [srt] get[x]{
	return [x];
	}
	public void set[x]([srt] [x]){
	this.[x] = [x];
	}

Function(x,arg,ret,b)	public [ret] [x](){
	[b]
	}

表 3-3 前三行是 IDSL 中的代码 Entity(x,body)会根据 SSH 框架转化为三种不同的类：Bean、DAO 和 Service。采用[x]来保证 Entity 的名字能准确的专递给 Bean、DAO 和 Service。PropertyNoAnno(x,propkind,srt)表示 Entity 中的 Property，它们会转化为 bean 中的属性，同时还有对应的 setter/getter 方法。Function(x,arg,ret,b)是用户定义的方法，其重写规则比较简单，将对应的部分填写到对应的 Service 中即可。详细的 Entity 的重写规则可以参见附录一。

2) View 的重写规则

根据 SSH 框架的结构，转换过程中需将 View 中的页面标签和 Action 分离操作分离，将页面标签分离到 jsp 文件中，将 Action 操作分离到 Action 中。同时对于页面标签，会同时包含 html 标签，jsp 标签以及 Struts 标签。这些必须根据情况预定义好。此外，页面和后台 Action 操作的相互绑定操作——即 form 表单提交所对应 Action 类必须定义好且采用“类名!方法名”的方式。由于 View 重写规则较为复杂，不仅仅是页面语法众多，语义复杂，同时 View 的重写规则不仅是包含页面，也包含后台逻辑。表 3-4 展示了一些具体的转换规则：

表 3-4 IDSL View 的抽象语法树到 Java 代码的转化规则

Table 3-4 Rewrite Rule from IDSL View Abstract Tree to Java Code

View 的抽象语法树	JSP 或 Java 语句
PageType(x,PageTile, PageFrame,body)	<%@page contentType="text/html;charset=utf-8" %> <%@ include file="/common/taglibs.jsp" %> <html><body> <%@ include file="[pageFrame]header.jsp" %> <%@ include file="[pageFrame]leftbar.jsp" %> <%@ include file="[pageFrame]content.jsp" %> <%@ include file="[pageFrame]footer.jsp" %> </body></html>

表 3-4 （续）

FormA(name,data,column, action)	split-action: FormA(name, data, column, action)->$[[jspContent]] with jspContent := <elem-to-html>FormA(name, data, column, action); ActionName := <guarantee-extension(|"java")><concat-strings>[name, "Action"]; ActionContent := <elem-to-action>FormA(name, data, column, action);<write-to-java-file>(ActionName, ActionContent);
FormA(name, data, column, action)	<s:form action="[name]Action.action" method="post"> <table width="100%" border="0" valign="top" cellpadding="0" cellspacing="0"> <tr> <td height="50" align="center" valign="middle" colspan="5"> <s:actionerror/> </td> </tr> <tr> <td colspan="5"> </td> </tr> [content] <tr> <td> </td> <td> </td> <td height="30" valign="middle"> 验证码 </td> <td> <input type="text" name="securityCode" class="textbox" size="4" autocomplete="off" /> </td> <td> </td> </tr> <tr align="center" valign="middle"> <td> </td> <td height="30" colspan="4" align="center">

表 3-4 （续）

	<input type="submit" class="unnamed1" onclick="validateRequired(loginForm)" value="登陆"> <input type="reset" class="unnamed1" value="重置"> </td> <td width="20%"> </td> </tr> <tr><td colspan="5"> </td></tr> </table> </s:form>

FormA(name,data, column, action)	@Results({ @Result(name = StrutsAction.RELOAD, location = [name]Action.URL, type = StrutsAction.REDIRECT) })
	public class [name]Action extends StrutsAction<[entity]>{
	public static final String URL = [name]Action".action";
	@Autowired
	private [entity]Service [entity']Service;
	
	@Override
	public String execute() throws Exception {
	[entity] [entity'] = ([entity])ActionContext.getContext().getSession().get("[entity']");
	[b]
	return SUCCESS;
	}
	}

表中 PageType 是 Page 的抽象语法树的一种表现形式，它包含了名字 x，title，PageFrame 和具体的 Page body。名字，title 这些内容是可以通过 PageType 的参数来获取，但 PageFrame 无法通过 PageType 来简单地获取，它必须通过对应的 PageFrame 文件中的内容才能获取。但鉴于 Spoofax 中文件操作功能不是很强大，无法做到便捷地做到对一个已存在的文件添加新内容，故本文采用的是 jsp:include 进行页面布局的方式合成 jsp 页面文件。采用 jsp:include 的好处在于整个重写规则可以顺序执行，而无需再执行过程中中断去执行另一段重写规则。而且 jsp:include 的页面布局方式是在页面上进行布局，而不是像 Struts 的 tiles 需要借助 XML 文件，这也使得制定重写规则的时间更简单。

表格中有三个 Form(name,data, column, action)，三者对应的是同一棵子树，但需要转化的内容不同，可以注意到其实第二个 Form 的重写规则就是第一个

Form 重写规则中的 elem-to-html 规则，第三个 Form 的重写规则就是第一个 Form 重写规则中的 elem-to-action 规则。第一个 Form 重写规则 split-action 可以分为两部分，一个是调用 elem-to-html 规则生成 jsp 页面，另一个就是调用 elem-to-action 生成 Action 类。

elem-to-html 规则比较简单，除了 Form 内容需要进一步重写，其它部分的 HTML 标签只要直接显示即可。这里 Form 生成时添加了验证码这一块，是在 Action 提交按钮之后。用户填好后会调用 Security/SecurityCodeImage.action 这个 Action 类来验证输入的验证码是否正确，这个类是预先定义好的类。

elem-to-action 规则需要将默认的 Action 操作 save，update，delete，search 转化为后台的 Action 类。首先生成的 Action 类是要继承自 StrutsAction 这个方法，因为 StrutsAction 已经包含了 save，update，delete，search 四种方法，生成的 Action 类只要简单的在 execute 方法中调用一下即可。然后生成的 Action 类需要泛型类，因为 StrutsAction 类需要泛型。之所以使用泛型是希望通过泛型了解整个 Action 会做的操作，将它的一些操作，如 Service 类都加载进来。这种对于 View 中默认的 Action 操作是比较容易知道的，因为 Form 表单会有数据绑定域 data。通过分析这个域，重写规则就能提取出其中所涉及的 Entity 类，用它对 Action 类进行泛型。部分的 View 规则可见附录二。

3) Action 的重写规则

根据 SSH 框架的结构，Action 直接转化为 Struts 的 Action 类，其中所有的 Action 都必须继承自 StrutsAction 类。表 3-5 显示了 IDSL Action 到 Struts Action 的一部分转化规则。

表 3-5 IDSL Action 的抽象语法树到 Java 代码的转化规则

Table 3-5 Rewrite Rule from IDSL Action Abstract Tree to Java Code

IDSL Action 的抽象语法树	Java 语句

Action(x,b)	[public class [x]Action extends ActionSupport implements Preparable{
	public String execute()
	[b']
	}

表中 Action 的重写规则叫 Form 中的简单，因为默认的增删改查用户可以通过 View 中默认的四个操作解决。增删改查四种操作最终用户用户也可以通过 Entity 带的方法来实现。自定义 Action 的编程相较 Entity 和 View 来说是比较困难的，最终用户如果能掌握这部分内容，则不需要 IDSL 提供复杂冗余的操作了。

4) 配置文件的重写规则

由于之前本文分别采用了注解方式代替 Hibernate 配置文件，默认的 Struts 配置文件，以及 AOP 注入方式写 Spring 配置文件。这些配置文件只要在工程一开始指定就行，之后就不需要修改了。

3.4 关键问题及其解决方案

本节主要研究使用 Stratego 进行 IDSL 语言编写重写规则时所遇到的一些关键问题，并提出了相应的解决方案。

3.4.1 基于内容的转换

在 IDSL 的语法中，当用户定义了 Entity 时，会同时将这个 Entity 相关的操作也定义在其中。而 SSH 要求用于持久化的 Entity 仅包含其属性，操作则在另外的 Service 中，交由 Spring 框架代理。因此，在此处转换中，需要从抽象语法树中，按照属性和方法的不同将其分离。

从 Entity 的抽象语法树 Entity(x,body)中，针对 body，编写重写规则如下：

```
split-entity-body: entbodydecs -> (props,functions)
```

这个规则将 body 内容分离成 props 和 functions 两部分。在分离过程中，首先应编写规则识别属性与方法的抽象语法树。对于属性，由于属性的抽象语法树不唯一，因此需要使用连接多个重写规则，形成重写策略，具体如下：

```
get-property-def =
        \Property(x,kind,srt,_) -> (x,kind,srt)\
        <+  \PropertyNoAnno(x,kind,srt) -> (x,kind,srt)\
        <+  \DerivedProperty(x,kind,srt,_,_) -> (x,kind,srt)\
        <+  \DerivedPropertyNoAnno(x,kind,srt,_) -> (x,kind,srt)\
is-property-cons = where(get-property-def)
```

而对于方法，其抽象语法树唯一，为 Function(_,_,_,_)，因此，编写重写规则如下：

```
is-function-cons = ?Function(_,_,_,_)
```

最后，借助 stratego 中提供的 filter 规则[28]，对整个 body 进行过滤，提取出 Properties 和 Functions，完整的规则如下：

```
split-entity-body :
    entbodydecs -> (props,functions)
    with
        props := <filter(is-property-cons)> entbodydecs
            ; functions := <filter(is-function-cons)> entbodydecs
```

3.4.2 容器类的遍历

在 Stratego 中，容器类的遍历是基于一个单步子项的递归闭包，即通过从递归式中分离出一个单步子操作，便可以定义很多遍历操作。

基于以上考虑，对于列表的递归遍历规则如下：

```
map(s):[] - > []
map(s):[x | xs] -> [<s>x|<map(s)>xs]
```

其中[]代表一个空的列表，[x|xs]代表一个非空列表，通过递归，可以将重写规则 s 应用于列表的每一项。

递归遍历广泛应用于闭包的转换，如当 Module 下有多个 Entity 定义，Entity 下有多个 Property 和 Function 时，通过递归遍历，可将闭包内的每个项都进行转换，具体转换规则如下：

```
to-entity:
                    t* -> <map(to-entity)> t*
```

3.4.3 列表下标取值的模拟

上一节中对列表元素的遍历是一种比较特殊的情况，即对容器中所有的元素采用统一的规则来进行重写。但对于从容器中根据下标取出某个元素无论是 Stratego 的使用说明还是 API 都没有提供，故本文根据其提供的 API 封装了一个取固定下标元素的操作：

```
gain：
(list, index)->elem
with
    elem:=<last><take(|index)>list
```

其中<take(|n)>这个规则是对一个 list 列表取前 n 个元素，这个操作取出来的还是一个列表。last 这个操作是从一个 list 列表中取出最后一个元素。

3.4.4 条件操作的使用

Stratego 中有提供条件操作 if(Strategy c, Strategy b)和 if(Strategy c, Strategy b1, Strategy b2)。前者表示 c 策略成功，则执行 b 策略，后者表示 c 策略成功执行 b1 策略，若失败则执行 b2 策略。但这个操作并不是由作者提供的且在 API 中没有任何说明和例子，在实际使用中也是存在问题，并不能实现条件跳转。本文采用的是 Stratego 自带的条件操作:

a<b+c

if(Strategy c, Strategy b)相当于 c<b+id，而 if(Strategy c, Strategy b1, Strategy b2)则相当于 c<b1+b2，其中 id 是 Stratego 的一个默认的符号，表示成功，与之对应的是 fail 表示失败。虽然 Stratego 自带的条件操作能够执行，但在使用时也会出现问题，条件操作只能放在 with 声明的最后一行，如果不是放在最后一行，则其余声明无法实现。为了解决这个问题，本文将所有会在整个条件语句操作之后执行的语句封装成规则，在条件语句中执行。

3.4.5 转换文件的生成

Stratego 提供了很多文件操作规则，但大都较为零散，且缺乏相关操作说明，较难使用。在本文中，文件的写操作会被大量使用，故为方便实际使用，本文将文件操作作了二次封装，以文件类型（Java，jsp，xml）为标准，封装了各种文件类型的写文件操作。以写 Java 类型文件为例，编写规则如下：

```
write-to-java-file:
    (filename,filecontent) -> None()
    with
        name := <guarantee-extension(|"java")>filename;
        handle := <fopen> (name, "w");
        <fputs> (filecontent, handle);
        Fclose
```

其中 guarantee-extension 这一规则用于对文件名的后缀名进行检查，保证文件的后缀名是 java 而不是其他类型。

3.5 本章小结

本章首先确定了三层领域特定语言到 Java 代码的总体设计思路，即将 SIDSL 转化为 UIDSL 语言，UIDSL 与 IDSL 语言使用同一个代码生成器。对于 IDSL 的

代码生成器，本章总结了程序转换领域已有的成果，并分析比较了各种技术的使用场景，选择了基于重写规则的程序转换技术。本章继而比较了已有的基于重写规则的程序转换框架，并设计了基于的 Stratego/XT 框架和 Spoofax 插件的 IDSL 到 Java 代码生成工具的架构。在此基础上，本文从四个方面分析现有的 SSH 框架的结构，结合 Spoofax 工具的特定，确定了代码重写规则的一些实施细节，并对这些细节做了相关的举例和说明。之后本节罗列出了一些在代码重写规则中的关键技术，并对这些技术做了研究和讨论。

第四章 实验

本章设计三个实验对面向最终用户的领域特定语言进行实验，实验案例为高校信息系统中的学生选课子系统。

第一个实验为 IDSL 实验，用以验证 IDSL 能支持最终用户进行编程，编写的程序能成功生成 Java 代码，同时对 IDSL 与 Java 的开发生产率做比较。

第二个实验为 UIDSL 实验，用以验证 UIDSL 比 IDSL 编码更快。

第三个实验为 SIDSL 实验，用以验证 SIDSL 的生产率比 UIDSL 高，编程更便捷。

4.1 IDSL 实验

4.1.1 实验过程

本实验邀请了一位领域专家去写 IDSL 代码，一位程序员编写 Java 代码。所编写的模块是选课模块，需求与第二章中所提及的需求一致。比较两者之间培训时间、工作量。之后用 IDSL 的代码生成工具将 IDSL 语言生成 Java 代码，将生成的 Java 代码量和 IDSL 代码量做比对。最后还会比较生成的 Java 的修改率。

4.1.2 实验结果与评估

1）工作量评估

采用 IDSL 编写代码和 Java 编写代码培训时间及实验程序工作量如表 4-1 所示。表中领域专家对 IDSL 的培训时间是 2 天（一天 8 小时算），开发相关的选课系统是 0.5 天，而程序员编写选课系统的时间是 2 天。虽然从总体上看，总的开发时间 IDSL 不比 Java 快，因为 IDSL 还有培训时间，但开发时间 IDSL 远比 Java 快。一旦一个领域专家熟悉了 IDSL 语言后，它的开发效率将比 Java 快。

表 4-1 IDSL 和 Java 之间培训时间和开发时间对比

Table 4-1 Comparison between IDSL and Java in Training Time and Coding Time

语言	培训时间（天）	开发时间（天）
IDSL	2 天	0.5 天
Java	0 天	2 天

2）代码生成

使用 IDSL 工具将其转化为对应的 Java 语言，两者对比如表 4-2 所示。本文选取了代码行数衡量的标准：

表 4-2 IDSL 和生成 Java 代码对比

Table 4-2 Comparison between IDSL and Generated Java Code

IDSL 文件		生成文件	
源文件名称	行数(LOC)	生成文件名称	行数(LOC)
IDSL	174	CourseAction.java	102
		Course.jsp	166
		basicFramehead.jsp	7
		basicFrameleftbar.jsp	52
		basicFramefooter.jsp	12
		loginFramehead.jsp	7
		loginFramefooter.jsp	12
		loginAction	43
		logoutAction	11
		User.java	52
		UserDao.java	9
		UserService.java	9
		Course.java	63
		CourseDao.java	11
		CourseService.java	25
		ClassType.java	68
		Schedule.java	61
		ScheduleDao.java	19
		ScheduleService.java	36
总共	178		765

表 4-2 中统计了 IDSL 编码时所涉及到的代码与最终生成的代码进行比对。经对比，发现 Java 的代码行数是 UIDSL 的 4.3 倍。虽然通过 IDSL 生成相应的 Java EE 工程在代码上会有一定的冗余度，但表中尚未统计 Java EE 工程中配置文件的行数，配置文件在 SSH 工程中也是有一定工作量的。此外 IDSL 的代码通过回车来分隔的语句而不是分号，基本上 IDSL 代码中不会出现一句很长的语句，以符合一页原则，所以总体而言 IDSL 的代码总量是原 Java 代码的四分之一或者

更少些。生成的 Java 代码运行效果图如图 4-1 所示：

图 4-1 IDSL 的运行结果

Fig. 4-1 Running of IDSL Codes

3）生成代码修改率

表 4-3 生成代码修改率

Table 4-3 Modification Rate for Generated Code

修改部分	修改数（行）	修改率（%）
Entity	49	11
View	42	10.5
Action	6	10
总共	97	11.2

表 4-3 显示了生成代码的修改率，很多修改率都是因为要添加 import 语句引起的。表中 Entity 的修改率最高，主要是 Entity 生成的类比较多，一个 Entity 生成三个 Java 类，需要三个地方添加 import 语句。而 View 的修改率相对低是因为页面内容多，pageFrame 基本不用修改。但 View 中默认 Action 操作转化为 Java 类时还是需要修改的，包括 import 语句、变量获取等。其实除了代码修改以外，工程配置文件的数据库配置其实是需要手动添加的，程序无法获知数据库的配置。

通过实验，得到了一些初步的结论：

1）IDSL 语言较为简单，能完成 Java EE 工程中一些基本内容的开发，同时使用 IDSL 语言进行开发，开发时间、代码量等方面都是优于普通编程语言的。

2）IDSL 的预定义特性在实际开发过程中是比较实用的，它能有效地减轻开

发者的负担。

3）IDSL View 中几种标签在一般的开发中能满足 Java EE 工程的需求。而且，这些标签中的默认 Action 操作能完成不少场景下的需要。

除了上述一些优点外，本文也发现 IDSL 的一些不足：

1）在 IDSL 的重写规则中，没有提供 import 等语句。因为这概念对最终用户而言有些陌生甚至费解，用户也不知道 import Java 中那个包。这是要代码生成好后再添加的。

2）虽然 IDSL 在代码转化方面是根据框架来实现，尽可能的降低代码重写的难度，但生成的代码还是可能存在一些错误，需要人工修改。

3）数据库必须事先确定，不能让最终用户去选择，因为这些都是在 applicationContext.xml 这个文件中定义好的。包括数据库用的 JDBC/ODBC 驱动包都是要预先下好，放到工程下。

4）Action 语言对最终用户而言难度较 Entity 和 View 大。Action 的语法比较接近 Java 的语法，对最终用户而言，无论是变量定义还是分支循环语句的使用远没有 Entity 和 View 提供的操作直观。他们更希望使用 View 中提供的增删改查这些默认的 Action 或者说是 Entity 中提供的默认方法。

4.2 UIDSL 实验

4.2.1 实验过程

本节邀请了一位领域专家用 UIDSL 代码编写 4.1 节中的选课子系统，并和 4.1 节中的 IDSL 作比较。

4.2.2 实验结果与评估

本节同样采用培训时间和工作量作为比较标准。比较结果如表 4-3 所示。

表 4-4 UIDSL 和 IDSL 之间培训时间和工作量比较

Table 4-4 Comparison between UIDSL and IDSLin Training Time and Coding Time

语言	培训时间（天）	开发时间（小时）
IDSL	2	4
UIDSL	2	2.5

通过实验，得到了初步的结论：UIDSL 和 IDSL 在语法上基本一致，故培训时间基本是一样的，但 UIDSL 编程比 IDSL 编程更快，这是由于 UIDSL 提供了

一些预定义的 Entity，View 和 Action，使得最终用户编码更加容易。此外，UIDSL 中预定义的 Entity，View 和 Action 在重写规则中是直接转化为对应的 Java 代码，这也会提高代码转化的成功率。

4.3 SIDSL 实验

4.3.1 实验过程

本节邀请了一位领域专家用 SIDSL 代码编写 4.1 节中的选课子系统，并和 4.2 节中的 UIDSL 作比较。

4.3.2 实验结果与评估

本节同样采用培训时间和工作量作为比较标准。比较结果如表 4-5 所示。

表 4-5 SIDSL 和 UIDSL 之间培训时间和工作量比较

Table 4-5 Comparison between SIDSL and UIDSLin Training Time and coding Time

语言	培训时间	开发时间
UIDSL	2 天	2.5 小时
SIDSL	0.5 天	1 小时

通过实验，得到了初步的结论：在选课领域，使用 SIDSL 编程比 UIDSL 编程更快，培训时间更短，更加面向最终用户。

最终用户在使用了相关语言编程后，认为三层语言让领域专家能参与编程，极大地发挥了他们的特长，此外都比较容易上手，能让最终用户处理以前程序员才能处理的任务。最终用户认为图形化编程的直观度会比文本编程更好。

4.4 本章小结

本章做了三个个简单的实验。本章选择了一个相对固定的领域作为实验的对象——选课子系统。第一个实验是用 IDSL 编程，并使用转换工具产生 Java 代码对两者进行了实验评估。该实验先将 IDSL 编码生产率和 Java 代码生产率做对比，对比了两者的培训时间和工作量，之后比对了 IDSL 的代码量和生成的 Java 的代码量，最后统计了生成代码的修改率。通过实验，本文可以得出结论 IDSL 基本可以定义一些简单的 Java EE 工程，具备一定的领域描述能力。其代码量和开发时间都比普通开发语言 Java 少。但转化的代码在细节上还是有不足的，需要最后用户去修改。第二个实验则是用 UIDSL 去编程代码，与第一个实验中的

IDSL 做对比。实验结果表明 UIDSL 比 IDSL 更为方便。第三个实验则是用 SIDSL 去编程代码，与第二个实验中的 UIDSL 做对比。实验结果表明 SIDSL 比 UIDSL 更为方便。至此可以得出结论：

1）IDSL 能支持最终用户进行编程，编写的程序能成功生成 Java 代码，并进行运行，且生产率比 Java 较高。

2）UIDSL 的编程比 IDSL 编程更快。

3）SIDSL 的抽象层次更高，最终用户更容易上手，生产率更高。

第五章 结束语

5.1 主要工作及总结

本文针对高校信息系统的现状，结合最新的领域特定语言和最终用户编程技术，提出了一套适用于信息系统的面向最终用户的领域特定语言。同时结合 Eclipse 插件 Spoofax，开发了一套 Java 代码生成工具。

本文的贡献可以归纳为以下几点：

1）提出了针对高校信息系统一套领域特定语言

本文分析并研究了高校信息系统的特定，结合领域特定语言和最终用户编程技术，提出了一套适用于高校信息系统的 DSL 语言。高校信息系统因业务繁多，逻辑复杂导致单以单层领域特定语言无法同时兼顾到面向最终用户和语言功能强大两个方面。本文以分层地思想对高校信息系统来定义领域特定语言，较好地解决了这一问题。三层语言分别是信息系统层，高校信息系统层和信息系统子领域层。三层语言所适用的领域逐渐缩小，语言也变得不断简洁、方便、面向最终用户。本文不仅给出了三层语言各自的语言－－IDSL、UIDSL 和 SIDSL，同时也提出了设计这三层语言的方法论。

2）开发了领域特定语言到 Java 代码生成工具

本文比较并分析了业界中代码生成技术的优缺点，并根据高校信息系统输入模型复杂和逻辑可变性高这些特点，选取了以重写规则为基础技术和 Spoofax 为开发工具，开发了 I2J、U2J、S2U 三个工具，实现了到 Java 的代码生成。本文借鉴业界比较成熟的 Web 应用程序开源框架——SSH 框架，并结合自身语言特点提出了一套简单可行的代码重写规则，同时本文针对代码重写规则的难点以及 Spoofax 的不足，提出了解决方案。

3）通过实验验证了语言的可行性和易用性

本文最后通过实验，比对了用 UIDSL 和 Java 分别开发同样一个选课系统所用的培训时间和工作开发时间，证实了 UIDSL 能使领域专家快速开发简易的高校信息系统功能。实验还对比了 SIDSL 和 IDSL，证实了 SIDSL 在选课子领域比 IDSL 更加方便简洁。

与传统的软件开发技术相比，使用面向最终用户的领域特定语开发高校信息系统具有以下几个优点：

1）有效减少开发周期。与专业开发人员不同，领域专家不需要通过需求调研，界面原型等步骤去反复确认开发需求，而向最终用户的领域特定语开发时间不逊于传统软件开发，整体开发周期将会明显缩短。

2）有效降低返工风险。不需要反复确认需求，避免了需求确认时的误解或歧义，提高了软件开发的质量。

3）有效降低维护成本。高校信息系统的领域专家远远多于传统软件开发人员，让领域专家去维护系统，能有效的节约开销，降低维护成本。

和现有相关研究工作相比，本文的研究工作具有以下优点：

和 WebDSL 相比，本文的领域特定语言是面向最终用户的，其词法、语法都比较简单。而 WebDSL 则是面向编程人员的，很多词法、语法对最终用户而言比较陌生艰涩。

和 OK 语言相比，本文的领域特定语言是针对高校信息系统的，无论是第二层的 UIDSL 的预定义特性还是第三层的选课子领域 SIDSL，都是方便最终用户编程的。而 OK 是面向管理信息系统的，语言的定义对于高校信息系统而言是不够的。

和其他面向最终用户的 DSL 相比，本文的领域特定语言是针对高校信息系统的一套面向最终用户的 DSL。目前的 DSL 中还没有类似的尝试或研究。

5.2 研究展望

由于时间仓促，本研究在许多地方还存在不足，有待改进和提高：

1）在代码转化方面，转化的 Java 往往会存在语法错误或运行时错误。这是由于 Spoofax 中变量申明、传递以及取值没有提供很好的操作，同时本文设计的重写规则尚不成熟，需要更多的实验去改进。

2）在代码转化方面，转化好的 Java 代码仍然需要人工手动修改，比如 import 语句，指定数据库等。这是因为面向最终用户是的 IDSL 与 Java 语法存在较大，无法完美的一一对应。这也需要更多的实验去改进。

3）在 Action 语法定义上比较接近 Java 语法，使得最终用户不易上手。信息系统有业务逻辑复杂多变，无法用一个比较高抽象层次的语言去概括。如果对信息系统能有更深入的了解，有大量开发经验，则能提出一个更好更抽象的语言。

4）第三层语言定义，选取样本过少，可能导致归纳的需求不完整，语言不够强大。同时对于所选样本，采用的是实际使用和查看一流系统代码，没有去了

解相关的需求文档，也可能导致归纳的需求不完整，语言不够强大。

参 考 文 献

[1] Martin Fowler，Domain-Specific Languages[M], Addison-Wesley Professional. 2010

[2] H. Lieberman, F. Paternó, M. Klann, V. Wulf, End-User Development: an Emerging Paradigm[M] , Springer. 2006

[3] Sumit Gulwani, Dimensions in Program Synthesis[C], PPDP Hagenberg, Austria. 2010

[4] Greg Little, Robert C. Miller, Victoria N. Chou, Michael Bernstein, Allen Cypher, Tessa Lau. Sloppy Programming[M]. MIT Computer Science and Artificial Intelligence Laboratory,IBM Almaden Research Center.2012.

[5] Greg Little and Robert C. Miller, Keyword Programming in Java[C], ASE , Atlanta, Georgia, USA.2007

[6] Vu le, Sumit Gulwani, Zhendong Su, SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language [C]. MobiSys. 2013

[7] David Bruce, What make a good domain-specific language[C], British Crown Copyright/DERA. 1997

[8] Zef HEMEL, Methods and Techniques for the Design and Implementation of Domain-Specific Languages[M], Printed and bound in The Netherlands by CPI W ¨ohrmann Print Service. 2012

[9] Debasish Ghosh, DSLs in Action[M], Manning Publications. 2011

[10] Martin Fowler, Language Workbeched: The Killer-App for Domain Specific Languages[EB/OL]
http://martinfowler.com/articles/languageWorkbench.html. 2010

[11] Frederic P. Miller, Agnes F. Vandome, John McBrewster, Intentional Programming[M], International Book Marketing Service Ltd. 2010

[12] Vytautas Stuikys, Robertas Damasevicius, Meta-Programming and Model-Driven Meta-Program Development: Principles, Processes and Techniques[M], Springer. 2013

[13] Paul Hudak, Build Domain-specific Embedded language[J]. Journal ACM Computing Surveys, Volume 28, 1996

[14] Paul Laird, Stephen Barrett, Towards Context Sensitive Domain Specific Languages[C], CAMS, June 16, Dublin, Ireland.2009

[15] Paul Klint, Tijs van der Storm, Jurgen Vinju, RASCAL: a Domain Specific Language for Source Code Analysis and Manipulation[C], SCAM. 2009

[16] Arie van Deursen Paul Klint Joost Visser, Domain-Specific Languages: An Annotated Bibliography[C], Dutch Telematica Instituut, project DSL.2006

[17] John Hughes , Why Functional Programming Matters, in Research Topics in Functional Programming[M], Addison-Wesley, 1990

[18] Sebastian Gunther，Agile DSL-Engineering with Patterns in Ruby[R], Technical Report FIN-08-2009.2009

[19] Anneke Kleppe， Software Language Engineering: Creating Domain-Specific Languages Using Metamodels[M]，Addison-Wesley Professional. 2008

[20] Eelco Visser, WebDSL: A Case Study in Domain-Specific Language Engineering[C], Generative and Transformational Techniques in Software Engineering II. 2008

[21] Danny M. Groenewegen , Zef Hemel, Lennart C.L. Kats , Eelco Visser, WebDSL: a domain-specific language for dynamic web applications[C], OOPSLA Companion to the 23rd ACM SIGPLAN conference. 2008

[22] Zef Hemel , Eelco Visser Declaratively Programming the Mobile Web with Mobl[C] Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications . 2011

[23] Martin Bravenboer, Eelco Visser Concrete Syntax for Object- Domain-Specific Language Embedding and Assimilation without Restrictions[C], OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada. 2004

[24] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, Patrick Valduriez, Model-based DSL Frameworks[C], OOPSLA. 2006

[25] Alen, OK QQ group[EB/OL] , http://qun.qzone.qq.com/group#!/115349254/share. 2013

[26] Visser E. Stratego/XT[EB/OL], http://hydra.nixos.org/build/4325353/download/1/manual/chunk-chapter/tutorial.html. 2008

[27] Bravenboer M, Kalleberg K T, Vermaas R, et al. Stratego/XT 0.17. A language and toolset for program transformation[J]. Science of Computer Programming, 2008, Volume 72 Issue 1: 52-70.

[28] Visser E. Program transformation with Stratego/XT[M],Domain-Specific Program Generation. Springer Berlin Heidelberg, 2004: 216-238.

[29] Olmos K, Visser E. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules[C],Compiler Construction. Springer Berlin Heidelberg, 2005: 204-220.

[30] De Jonge M, Visser E, Visser J. XT: A bundle of program transformation tools system description[J]. Electronic Notes in Theoretical Computer Science, 2001,

Volume 44 Issue 2: 79-86.

[31] IBM 基于插件的体系结构[EB/OL], http://www.ibm.com/developerworks/cn/java/os-ecplug/index.html, 2003

[32] Kalleberg K T, Visser E. Spoofax: An Extensible, Interactive Development Environment for Program Transformation with Stratego/XT[C]. Software Engineering Research Group 2007.

[33] Bruce Eckel. Java编程思想[M]. 北京：机械工业出版社，2007. 313~351页

致　谢

在硕士论文即将完成之际，我想借此机会对每个关心过我，给过我帮助的老师、同学、朋友以及亲人给予衷心的感谢和崇高的致敬。

首先我要感谢我的导师——沈备军老师。无论是我开题阶段、研究阶段还是撰写硕士论文阶段，沈老师每周都会关心我的研究进展，及时分析我研究中的不足，不断地匡正我的研究道路，使我在研究的道路上能事半功倍。同时沈老师治学严谨，对我严格要求，不断鞭策我，使我受益匪浅。沈老师知识渊博，思维缜密，其高尚的学术素养给我树立了一个崇高的模板。同时我也要感谢所有软件学院教过、执导过我的老师。

其次我要感谢贾颖、朱剑钢以及所有实验室中给予我帮助的同学们。与他们相处不但使我们相互学习，一起成长，同时也让我感受到了实验室的温暖。每逢佳节，实验室总会组织各种活动，这给我留下了美好的回忆。

最后我要感谢我父母，正是由于他们的付出，我才能有今天的成果。

感谢所有给予我关心、快乐，伴我一起成长的人，正是由于你们的存在，我硕士生涯才精彩，才能立体。

攻读硕士学位期间已发表的学术论文

[1] Yu Tianyu，Shen Beijun. An End-User Domain-Specific Language for University Information Systems，The 1st International Conference on Information System and Electronic Commerce（ICITEC 2013）, Harbin, China, Dec 28-29, 2013. 已录用

附录一 部分 Entity 重写规则

```
rules // Incremental code generation of project using compilation library.

  index-compile-ast(|file, subfile):
    ast -> None()
    with
      java := <to-entity> ast;
      full-path := <dirname> file;
      filename := <guarantee-extension(|"java")> <base-filename> file;
      writePath := $[[full-path]/];
      writeFile :=   $[[writePath][filename]];
      try(<mkdir> writePath);
      <fclose> <fputs> (java, <fopen> (writeFile, "w"))

rules // Transformation to java strings.

  to-entity:
    [_] -> <concat-strings> <map(to-entity)>

  to-entity:
    () -> ""


  to-entity:
    Module(x, def*) ->
    $[ package [x];
        [def*]
     ]
    with
      debug(!"befor definition!");
      def'* := <to-entity> def*;
      debug(!"after definition！ ")
  to-entity:

    Entity(x,body)->((EntityFileName,EntityContent),(FunctionFileName,FunctionContent),(DaoName,DaoContent))
  with
    (props,functions) := <split-entity-body>body;
    EntityFileName := <guarantee-extension(|"java")>x;
    debug(!EntityFileName);
    EntityContent := <build-entity-content>Entity(x,body);
    debug(!EntityContent);
```

```
FunctionFileName := <guarantee-extension(|"java")><concat-strings>[x,"Manager"];
debug(!FunctionFileName);
FunctionContent := <build-function-content>Entity(x,body);
debug(!FunctionContent);
DaoName := <guarantee-extension(|"java")><concat-strings>[x,"Dao"];
DaoContent := <build-dao-content>Entity(x,body);
<write-to-java-file>(EntityFileName,EntityContent);
<write-to-java-file>(FunctionFileName,FunctionContent);
<write-to-java-file>(DaoName,DaoContent)

build-entity-content:
Entity(x,body)->
$[    @Entity
      @Table(name = "[x]")
public class [x] implements java.io.Serializable {
      private Integer id;
            @Id
            @TableGenerator(
                  name="tab_stone",
                  table="generator_table",
                  pkColumnName = "g_key",
                  valueColumnName = "g_value",
                  pkColumnValue= "[x]_pk",
                  allocationSize=1
            )
            @GeneratedValue(strategy = GenerationType.TABLE,generator="tab_stone")
            public Integer getId() {
                  return id;
            }
            public void setId(Integer id) {
                  this.id = id;
            }

         [props']
    }
 ]
with
      (props,functions) := <split-entity-body>body;
      props' := <map(to-entity)>props

build-dao-content:
Entity(x,body)->
$[@Repository
public class [x]Dao extends HibernateDao<[x]>{
```

```
        [functions']
    }
    ]
    with
        debug(!"before split!");
        (props,functions) := <split-entity-body>body;
        functions' := <map(to-entity)>functions

build-function-content:
    Entity(x,body)->
    $[@Service
    public class [x]Manager{
        @Autowired
        private [x]Dao dao;
        [functions']
    }
    ]
    with
        debug(!"before split!");
        (props,functions) := <split-entity-body>body;
        functions' := <map(to-manager)>functions

to-manager:
    Function(x,arg,ret,b)->
    $[public [ret'] [x]([arg']){
        return dao.[x]();
    }
    ]
    with
        ret' := <to-type>ret;
        arg' := <expression-to-java>arg
/*  to-entity:
    Entity(x,body)->
    $[
        public class [x]{
            [props']
            [functions']
        }
    ]
    with
        debug(!"before split!");
        (props,functions) := <split-entity-body>body;
        props' := <map(to-entity)>props;
    functions' := <map(to-entity)>functions;
```

```
    result :=  $[[functions']];
    filename := "test1.java";
    <write-to-java-file>(filename,result)*/
to-entity:
    PropertyNoAnno(x,AnyProp(),srt)->
    $[
        private [srt'] [x];
        public [srt'] get[x']() {
            return [x];
        }

        public void set[x'] ([srt'] [x]) {
            this.[x] = [x];
        }
    ]
    with
        debug(!"--------------------Property!");
        srt' := <to-type>srt;
        x' := <capitalize-string>x

to-entity:
    PropertyNoAnno(x,Ref(),srt)->
    $[
        private [srt'] [x];
        @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "[x]_id")
        public [srt'] get[x']() {
            return [x];
        }

        public void set[x'] ([srt'] [x]) {
            this.[x] = [x];
        }
    ]
    with
        debug(!"--------------------Property!");
        srt' := <to-type>srt;
        x' := <capitalize-string>x

to-entity:
    Function(x,arg,ret,b)->
    $[public [ret'] [x]([arg'])
    [b']
    ]
```

```
    with
        ret' := <to-type>ret;
        b' := <function-to-java>b;
        arg' := <expression-to-java>arg

to-entity:
    t* -> <map(to-entity)> t*

    is-property-cons = where(get-property-def)

    is-function-cons = ?Function(_,_,_,_)
    // get (name, kind, sort) tuple for property constructor
    get-property-def =
                \Property(x,kind,srt,_) -> (x,kind,srt)\
        <+    \PropertyNoAnno(x,kind,srt) -> (x,kind,srt)\
        <+    \DerivedProperty(x,kind,srt,_,_) -> (x,kind,srt)\
        <+    \DerivedPropertyNoAnno(x,kind,srt,_) -> (x,kind,srt)\


split-entity-body :
    entbodydecs -> (props,functions)
    with
    props := <filter(is-property-cons)> entbodydecs
        ; functions := <filter(is-function-cons)> entbodydecs
```

附录二 部分 Form 重写规则

```
split-action:
        FormA(name, data, column, action)->$[[jspContent]]
    with
        jspContent := <elem-to-html>FormA(name, data, column, action);
        ActionName :=    <guarantee-extension(|"java")><concat-strings>[name, "Action"];
        ActionContent := <elem-to-action>FormA(name, data, column, action);
        PreActionName :=    <guarantee-extension(|"java")><concat-strings>[name, "PreAction"];
        PreActionContent := <elem-to-preaction>FormA(name, data, column, action);
        <write-to-java-file>(ActionName, ActionContent);
        <write-to-java-file>(PreActionName, PreActionContent)

    split-action:
        FormB(data, column, action)->$[[jspContent]]
    with
        jspContent := <elem-to-html>FormB(data, column, action);
        ActionName := <guarantee-extension(|"java")>"FormAmAction";
        ActionContent := <elem-to-action>FormB(data, column, action);
        PreActionName :=    <guarantee-extension(|"java")>"FormAmAction";
        PreActionContent := <elem-to-preaction>FormB(data, column, action);
        <write-to-java-file>(ActionName, ActionContent);
        <write-to-java-file>(PreActionName, PreActionContent)

    elem-to-html:
        FormA(name, data, column, action)->
        $[
            <s:form action="[name]Action.action" method="post">
            <table width="100%" border="0" valign="top" cellpadding="0" cellspacing="0">
            <tr>
            <td height="50" align="center" valign="middle" colspan="5">
                <font color="#FF0000"><s:actionerror/></font>
            </td>
            </tr>
            <tr>
                <td colspan="5"> </td>
            </tr>
            [content]
            <tr>
            <td> </td>
            <td> </td>
            <td height="30" valign="middle">
            <font size="2">验证码</font>
```

```
                </td>
                <td>
                    <input type="text" name="securityCode" class="textbox" size="4"
autocomplete="off" />
                    <img src="Security/SecurityCodeImage.action" border="0" alt="请输入此验证码，
如看不清请点击刷新" onclick="window.location.reload(true)" style="cursor:pointer" />
                </td>
                <td> </td>
                </tr>
                <tr align="center" valign="middle">
                <td> </td>
                <td height="30" colspan="4" align="center">
                <input type="submit" class="unnamed1" onclick="validateRequired(loginForm)"
value="登陆">
                       
            <input type="reset" class="unnamed1" value="重置">
                </td>
                <td width="20%"> </td>
                </tr>
                <tr><td colspan="5"> </td></tr>
                </table>
            </s:form>
            ]
            with
                content := <split-content> FormA(name, data, column, action);
                actionNametemp* := <gain-action-name> action
                debug(!<length>actionNametemp*)
                <eq>("none", actionNametemp) < actionName := <concat-strings>[name, "Action"] +
actionName := <take(|1)><gain-action-name> action

    elem-to-html:
        FormB(data, column, action)->
        $[
            <s:form action="[actionName].action" method="post">
                <table width="100%" border="0" valign="top" cellpadding="0" cellspacing="0">
            <tr>
            <td height="50" align="center" valign="middle" colspan="5">
                <font color="#FF0000"><s:actionerror/></font>
            </td>
            </tr>
            <tr>
                <td colspan="5"> </td>
            </tr>
            <tr>
```

```
                <td> </td>
                <td> </td>
                <td height="30" valign="middle">
            <font size="2">用户名</font>
                </td>
                <td width="30%">
                <input type="text" name="username" class="textbox">
                </td>
                <td width="20%"> </td>
                </tr>
                <tr>
                <td> </td>
                <td> </td>
                <td height="30" valign="middle">
                <font size="2">密码</font>
                </td>
                <td>
                <input type="password" name="password" class="textbox" autocomplete="off" />
                </td>
                <td width="20%"> </td>
            </tr>
            <tr>
            <td> </td>
            <td> </td>
            <td height="30" valign="middle">
            <font size="2">验证码</font>
            </td>
            <td>
            <input type="text" name="securityCode" class="textbox" size="4" autocomplete="off" />
            <img src="Security/SecurityCodeImage.action" border="0" alt="请输入此验证码，如看不
清请点击刷新" onclick="window.location.reload(true)" style="cursor:pointer" />
            </td>
            <td> </td>
            </tr>
            <tr align="center" valign="middle">
            <td> </td>
            <td height="30" colspan="4" align="center">
            <input type="submit" class="unnamed1" onclick="validateRequired(loginForm)" value="登
陆">
               
        <input type="reset" class="unnamed1" value="重置">
            </td>
            <td width="20%"> </td>
            </tr>
```

```
        <tr><td colspan="5"> </td></tr>
        </table>
            </s:form>
        ]
        with
            actionNametemp := <take(|1)><gain-action-name> action;
            <eq>("none", actionNametemp) < actionName := "FormAmAction" + actionName :=
<take(|1)><gain-action-name> action

    prop-to-html:
        EntityProperty(a, b*)->
        $[[a'][b'*]]
        with
            a' := <sort-fetch>a;
            b'* := <map(sort-fetch-plus)>b*

    prop-to-name:
        EntityProperty(a, b*)->
        $[[a'][b'*]]
        with
            a' := <sort-fetch>a;
            b'* := <map(sort-fetch)>b*

    prop-to-html:
        EntityProperty(SimpleSort(a), b*)->
        $[[a'][b'*]]
        with
            a' := <sort-fetch>a;
            b'* := <map(sort-fetch-plus)>b*;
            bb := <take(|1)><map(sort-fetch)>b*;
            name := <filter(where(<eq>("SuperLink",a')))>bb;
            nameAction := <filter(where(<string-ends-with(|"Action")> name))> name;
            namePage := <filter(where(<string-ends-with(|"Page")> name ))> name;
            debug(!name);
            debug(!namePage)
    sort-fetch:
        SimpleSort(content)->content

    sort-fetch-plus:
        SimpleSort(content)->content'
        with
            content' := <concat-strings>[".", content]

    superLink:
```

```
        String(content)->None()

    elem-to-action:
        FormA(name, data, column, action)->
        $[
        @Results({ @Result(name = StrutsAction.RELOAD, location = [name]Action.URL, type = StrutsAction.REDIRECT) })
        public class [name]Action extends StrutsAction<[entity]>{
            public static final String URL = [name]Action".action";
            @Autowired
            private [entity]Manager [entity']Manager;
            @Override
            public String execute() throws Exception {
                [precall]
                [entity] [entity'] = ([entity]) ActionContext.getContext().getSession().get("[entity']");
                return SUCCESS;
            }
        }
        ]
        with
            entity :=  <gain-entity-name>data;
            entity' := <uncapitalize-string> entity;
            <eq>("User", entity) < precall := $[] + precall := $[User user = (User) ActionContext.getContext().getSession().get("user");]


    elem-to-action:
        FormB(data, column, action)->
        $[
        @Results({ @Result(name = StrutsAction.RELOAD, location = FormAmAction.URL, type = StrutsAction.REDIRECT) })
        public class FormAmAction extends StrutsAction<[entity]>{
            public static final String URL = FormAm".action";
            @Autowired
            private [entity]Manager [entity']Manager;

            @Override
            public String execute() throws Exception {
                [precall]
                [entity] [entity'] = ([entity]) ActionContext.getContext().getSession().get("[entity']");
                return SUCCESS;
            }
```

```
        }
        ]
        with
            entity := <gain-entity-name>data;
            entity' := <uncapitalize-string> entity;
            <eq>("User", entity) < precall := $[] + precall := $[User user = (User) ActionContext.getContext().getSession().get("user");]

    elem-to-preaction:
        FormA(name, data, column, action)->
        $[
        @Results({ @Result(name = StrutsAction.RELOAD, location = [name]Action.URL, type = StrutsAction.REDIRECT) })
        public class [name]Action extends StrutsAction<[entity]>{
            public static final String URL = [name]Action".action";
            @Autowired
            private [entity]Manager [entity']Manager;

            @Override
            public String execute() throws Exception {
                [precall]
                [entity] [entity'] = ([entity]) ActionContext.getContext().getSession().get("[entity']");
                return SUCCESS;
            }
        }
        ]
        with
            entity := <gain-entity-name>data;
            entity' := <uncapitalize-string> entity;
            <eq>("User", entity) < precall := $[] + precall := $[User user = (User) ActionContext.getContext().getSession().get("user");]

    elem-to-preaction:
        FormB(data, column, action)->
        $[
        @Results({ @Result(name = StrutsAction.RELOAD, location = FormAmAction.URL, type = StrutsAction.REDIRECT) })
        public class FormAmAction extends StrutsAction<[entity]>{
            public static final String URL = FormAm".action";
            @Autowired
            private [entity]Manager [entity']Manager;

            @Override
```

```
            public String execute() throws Exception {
                [precall]
                [entity]  [entity']  =  ([entity])
ActionContext.getContext().getSession().get("[entity']");
                return SUCCESS;
            }
        }
        ]
        with
            entity :=  <gain-entity-name>data;
            entity' := <uncapitalize-string> entity;
            <eq>("User", entity) < precall := $[] + precall := $[User user = (User)
ActionContext.getContext().getSession().get("user");]

    split-content:
        FormA(name, DataSchemaTypeA(data*, sql), ColumnSchemaType(column*), action) ->
elem
        with

            list := <map(tran-column)> column*;
            elem := <concat-strings>list;
            debug(!elem)

    split-content:
        FormA(name, DataSchemaTypeB(data*), ColumnSchemaType(column*), action) ->
        $[<tr>
        [elem]
        </tr>]
        with
            column'* := <map(tran-column)> column*;
            data'* := <map(tran-action)> data*;
            elem := <zip(conc-strings)>(column'*,data'*);
            debug(!elem);
            debug(!column'*);
            debug(!data'*);
            count :=<new-counter>;//计数器
            length := <length>column*;
            table := <new-hashtable>;
            elem := $[];
            repeat(debug(!(<next-counter> count))|length);
            <new-counter> count;
            debug(!(<length>column*));
            debug(!<inc>(<get-counter> count));
            debug(!(<take(|<inc>(<get-counter> count))> column*));
```

```
                <hashtable-put(|"elem", "a")> table;
                <hashtable-push("elem", "b")> table;
                repeat(<next-counter>  count;<hashtable-put(|"elem",  (<take(|<get-counter>  count)>
column*))> table|length);
                for(l:=<length>column* , count , length);
                for(elem := $[] , 0, length);
                <reset-counter> count;
                debug(!<take(|1)>(<hashtable-get(|"elem")> table))

    split-content:
        FormB(data, column, action) -> elem
        with
            elem := None()

    gain-entity-name:
        DataSchemaTypeA(data*, sql) -> name
        with
            name := <gain-entity-name-sub>(<last>data*)

    gain-entity-name:
        DataSchemaTypeB(data*) -> name
        with
            name := <gain-entity-name-sub>(<last>data*)

    gain-entity-name-sub:
        DataBinding(str, operation, EntityProperty(a, b*)) -> a'
        with
            a' := <sort-fetch>a

    tran-column:
        ColumnSchemaContentA(str, type)->$[
                <td  bgcolor="#f0f0f0"  width="8%"  valign="middle"  align="right"
colspan="1"><b>[str]</b></td>
                <td bgcolor="#ffffff" width="15%" class="nav status" colspan="1">
            <input type="[typeName]" ]
         with
            typeName := <get-constructor>type
//              debug(!typeName)

    tran-column:
        ColumnSchemaContentB(str)->elem
        with
            elem := $[
                <td  bgcolor="#f0f0f0"  width="8%"  valign="middle"  align="right"
```

```
colspan="1"><b>[str]</b></td>
                    <td bgcolor="#ffffff" width="15%" class="nav status" colspan="1">
                <input type="input" ]

    tran-action:
        DataBinding(str, operation, property) ->
        $[name="messageID"   id="messageID"   value="${[prop']}"   class="noIndentTextEntry"
size="30"/>
        </td>]
        with
            prop' :=   <prop-to-html>property

    gain-action-name:
        ActionSchema(elem*) -> content*
        with
//          debug(!elem*);
            content* := <map(gain-action-name-sub)>elem*
//          content := <gain-action-name-sub><take(|2)>elem*;
//          debug(!content*)
//          content* := $["d"]

    gain-action-name-sub:
        ActionSchemaContent(str, type) -> name
        with
//          debug(!str);
            name := <actionType-to-name>type

    actionType-to-name:
        ok(ActionId) -> ActionId

    actionType-to-name:
        save() -> $["save"]

    actionType-to-name:
        delete() -> $["delete"]

    actionType-to-name:
        update() -> $["update"]

    actionType-to-name:
        search() -> $["search"]

    actionType-to-name:
        reset() -> $["reset"]
```

面向最终用户的领域特定语言的研究

作者：郁天宇
学位授予单位：上海交通大学

引用本文格式：郁天宇 面向最终用户的领域特定语言的研究[学位论文]硕士 2014