



دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

پروژه کارشناسی

گرایش نرم افزار

طراحی و پیاده سازی

نرم افزار ساخت نمودار کلاس و تبدیل و اضافه آن به کد زبان سی

نگارش

امیررضا شیرمست

استاد راهنما

دکتر محمدرضا رزازی

شهریور ۱۴۰۰

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

به نام خدا

تعهد نامه اصالت اثر

تاریخ: مهر

اینجانب امیررضا شیرمست متعهد می شوم که مطالب مندرج در این پایان نامه حاصل کار پژوهشی اینجانب تحت نظارت و راهنمایی اساتید دانشگاه صنعتی امیرکبیر بوده و به دستاوردهای دیگران که در این پژوهش از آنها استفاده شده است مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذکر گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم سطح یا بالاتر ارائه نگردیده است.

در صورت اثبات تخلف در هر زمان، مدرک تحصیلی صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پیگیری قانونی خواهد داشت.

کلیه نتایج و حقوق حاصل از این پایان نامه متعلق به دانشگاه صنعتی امیرکبیر می باشد. هرگونه استفاده از نتایج علمی و عملی، واگذاری اطلاعات به دیگران یا چاپ و تکثیر، نسخه برداری، ترجمه و اقتباس از این پایان نامه بدون موافقت کتبی دانشگاه صنعتی امیرکبیر ممنوع است. نقل مطالب با ذکر مآخذ بلامانع است.

امیررضا شیرمست

امضا

سپاسگزاری

اینجانب، امیررضا شیرمست از استاد راهنما خود، جناب آقای دکتر محمدرضا رزازی
جهت تمامی کمک‌هایی که در مسیر نوشتن این گزارش به من کردند، سپاس گزارم.

امیررضا شیرمست

مهر ۱۴۰۰

چکیده

در سال ۱۹۷۲ زبان سی در آزمایشگاه بل ساخته شد و از آن زمان نسخه های متعددی منتشر شده است. این زبان یک زبان رویه ای و ساخت یافته است. سپس ۷ سال بعد از آن در همان آزمایشگاه زبان سی پلاس پلاس که از ۲ زبان سی و سیمولا تاثیر گرفته بود ساخته شد که امکانات یک زبان شی گرا علاوه بر امکانات زبان سی را داراست. با وجود بیشتر بودن امکانات زبان سی پلاس پلاس، برنامه های زیادی هستند که به زبان سی نوشته می شوند که امکانات یک زبان شی گرا را پشتیبانی نمی کند. در این پروژه قرار است یک نرم افزار کمک کننده به توسعه ساخته شود که برنامه نویسی شی گرا در زبان سی را هموار می کند.

کاربرد نرم افزار در محیط های توسعه ایست که از زبان سی فقط می تواند استفاده بشود. (مانند طراحی سیستم عامل و رایانش بی درنگ) توسعه دهندگان تلاش کرده اند برنامه نویسی شی گرا را در این زبان انجام بدهند و در نتیجه کتاب ها و مقالات زیادی در این زمینه موجود است که به توسعه دهندگان کمک می کند که یک برنامه با منطق شی گرا در سی بنویسند. در نتیجه وجود این نرم افزار که قابلیت ارث بری و داشتن متد های عمومی و خصوصی و سازنده و تخریب گر و چند ریختی را به یک ساختمان داده می دهد بسیار مفید خواهد بود.

واژه های کلیدی:

مهندسی نرم افزار به کمک کامپیوتر، انتقال نرم افزار، برنامه نویسی شی گرا، زبان سی

فهرست مطالب

| | | |
|----|--|----|
| ۱ | مقدمه | ۱ |
| ۴ | ۲ مفاهیم پایه | ۴ |
| ۵ | ۱-۲ ویژگی های پروژه | ۵ |
| ۶ | ۲-۲ الگو های برنامه نویسی | ۶ |
| ۱۴ | ۳-۲ اصطلاحات مربوط به الگو های برنامه نویسی | ۱۴ |
| ۱۹ | ۴-۲ زبان سی | ۱۹ |
| ۲۷ | ۵-۲ نمودار کلاس | ۲۷ |
| ۳۰ | ۶-۲ تحلیل گری لغوی | ۳۰ |
| ۳۱ | ۳ کار های پیشین | ۳۱ |
| ۳۲ | ۱-۳ تبدیل نمودار کلاس به کد | ۳۲ |
| ۳۶ | ۲-۳ کامپایل سی پلاس پلاس به سی | ۳۶ |
| ۳۷ | ۳-۳ برنامه نویسی شی گرا در زبان سی | ۳۷ |
| ۳۸ | ۴ دلایل برنامه نویسی شی گرا در زبان سی | ۳۸ |
| ۳۹ | ۱-۴ دلایل استفاده از الگوی برنامه نویسی شی گرا | ۳۹ |
| ۴۰ | ۲-۴ دلایل استفاده از زبان سی | ۴۰ |
| ۴۱ | ۵ طراحی و پیاده سازی | ۴۱ |
| ۴۲ | ۱-۵ قابلیت های ارائه شده در نرم افزار | ۴۲ |
| ۴۴ | ۲-۵ ورودی نرم افزار و کارکرد آن | ۴۴ |
| ۴۶ | ۳-۵ ساختمان اطلاعات کلاس ها | ۴۶ |

| | | |
|-----|-------------------------------|----|
| ۵-۴ | خطاهای پایه | ۵۳ |
| ۵-۵ | وابستگی کلاس ها | ۵۶ |
| ۵-۶ | تولید کد ابتدایی در مرحله اول | ۶۱ |
| ۵-۷ | تولید کد نهایی در مرحله دوم | ۶۵ |
| ۶ | ارزیابی | ۶۶ |
| ۷ | نتیجه گیری | ۷۸ |
| ۷-۱ | جمع بندی | ۷۹ |
| ۷-۲ | چالش ها و کارهای آینده | ۷۹ |
| | منابع و مراجع | ۸۱ |

فهرست اشکال

- شکل ۱-۲: تغییر ترتیب ۲ خط ۷
- شکل ۲-۲: ارتباط برنامه نویسی دستوری، ساخت یافته و رویه ای با همدیگر ۱۰
- شکل ۳-۲: استفاده از زیرنوع به وسیله ارث بری ۱۳
- شکل ۴-۲: تابع minimum با دو ورودی با نوع int ۱۵
- شکل ۵-۲: تابع minimum با استفاده از قابلیت template در زبان سی پلاس پلاس ۱۵
- شکل ۶-۲: مثال چندریختی زیرگونه متد ها در کلاس به زبان C++ ۱۶
- شکل ۷-۲: نتیجه مثال چندریختی زیرگونه متد ها در کلاس به زبان سی پلاس پلاس ۱۷
- شکل ۸-۲: مثال سربار گذاری توابع به زبان سی پلاس پلاس ۱۸
- شکل ۹-۲: مثال ۲ کلاس که متد غالب شده دارند ۱۹
- شکل ۱۰-۲: تعریف ۲ ساختار X و Y ۲۰
- شکل ۱۱-۲: تعریف چند تابع با اسم یکسان print در سی پلاس پلاس ۲۰
- شکل ۱۲-۲: نتیجه تعریف چند با اسم یکسان در سی ۲۱
- شکل ۱۳-۲: تعریف چند تابع با اسم و امضای متفاوت در سی ۲۲
- شکل ۱۴-۲: توانایی فراخوانی چند تابع با اسم print با استفاده از پیش پردازنده ها ۲۳
- شکل ۱۵-۲: استفاده از کلیدواژه print در تابع main ۲۴
- شکل ۱۶-۲: استفاده از تابع توکار __builtin_types_compatible_p ۲۵
- شکل ۱۷-۲: قالب تعریف union ۲۶
- شکل ۱۸-۲: تعریف مشخصه ها در داخل کلاس ۲۷
- شکل ۱۹-۲: تعریف انجمن ها برای مشخصه ۲۸
- شکل ۲۰-۲: تعیین تعداد در نمودار کلاس ۲۸
- شکل ۲۱-۲: تعیین وراثت در نمودار کلاس ۲۹

| | |
|---|----|
| شکل ۲-۲۲: یک نمودار کلاس کامل..... | ۲۹ |
| شکل ۳-۱: یک نمودار کلاس ارائه شده در نرم افزار visual paradigm..... | ۳۲ |
| شکل ۳-۲: خروجی کد نمودار کلاس ارائه شده در نرم افزار visual paradigm..... | ۳۳ |
| شکل ۳-۳: دیاگرام نمونه در نرم افزار Atash..... | ۳۴ |
| شکل ۳-۴: فایل تعریف ساختار کلاس Bar..... | ۳۵ |
| شکل ۳-۵: فایل سرایند متد های کلاس Bar..... | ۳۵ |
| شکل ۳-۶: فایل تعریف متد های کلاس Bar..... | ۳۶ |
| شکل ۵-۱: نمودار گرفتن ورودی..... | ۴۴ |
| شکل ۵-۲: نمودار حلقه خروجی در حالت ورودی گرافیکی..... | ۴۵ |
| شکل ۵-۳: نمودار حلقه خروجی در حالت ورودی XML..... | ۴۶ |
| شکل ۵-۴: نمودار کلاس ساختمان اطلاعات کلاس ها..... | ۴۷ |
| شکل ۵-۵: نمودار وابستگی کلاس ها با حضور کلاس های محیط گرافیکی..... | ۴۸ |
| شکل ۵-۶: توابع واسط DescriptiveMember..... | ۴۸ |
| شکل ۵-۷: ورودی xml ساختار ValueType..... | ۴۹ |
| شکل ۵-۸: ورودی GUI ساختار ClassAttribute..... | ۵۰ |
| شکل ۵-۹: ورودی XML ساختار ClassAttribute..... | ۵۰ |
| شکل ۵-۱۰: ورودی XML ساختار ClassConstructor..... | ۵۱ |
| شکل ۵-۱۱: ورودی GUI ساختار ClassMethod..... | ۵۱ |
| شکل ۵-۱۲: ورودی XML ساختار ClassMethod..... | ۵۲ |
| شکل ۵-۱۳: ورودی XML ساختار ClassStructure..... | ۵۲ |
| شکل ۵-۱۴: ورودی GUI ساختار ClassDiagram..... | ۵۳ |
| شکل ۵-۱۵: ورودی GUI ساختار ClassDiagram..... | ۵۳ |
| شکل ۵-۱۶: الگوریتم پیدا کردن دور های وابستگی..... | ۶۰ |

- شکل ۵-۱۷: ساختمان برای تولید کد ۶۱
- شکل ۵-۱۸: ساختمان متد برای تولید کد ۶۲
- شکل ۵-۱۹: قالب ذخیره سازی ساختار یک کلاس ۶۲
- شکل ۵-۲۰: فایل تعریف برای پیاده سازی متدها ۶۳
- شکل ۵-۲۱: تعریف متدهای برای یک کلاس ۶۴
- شکل ۵-۲۲: پیاده سازی متدهای پدر و توابع تخصیص حافظه برای یک کلاس ۶۴
- شکل ۶-۱: نمودار آزمون اول ۶۹
- شکل ۶-۲: بخشی از کد خروجی مرحله اول آزمون اول ۶۹
- شکل ۶-۳: کد استفاده شده در آزمون اول ۷۰
- شکل ۶-۴: نمودار آزمون دوم ۷۰
- شکل ۶-۵: بخشی از کد خروجی سرآیند مرحله اول آزمون دوم ۷۱
- شکل ۶-۶: بخشی از کد خروجی پیاده سازی شده مرحله اول آزمون دوم ۷۲
- شکل ۶-۷: کد استفاده شده در آزمون دوم ۷۲
- شکل ۶-۸: نمودار آزمون سوم ۷۳
- شکل ۶-۹: بخشی از کد خروجی سرآیند مرحله اول آزمون دوم ۷۳
- شکل ۶-۱۰: بخشی از کد خروجی پیاده سازی شده مرحله اول آزمون سوم ۷۴
- شکل ۶-۱۱: کد استفاده شده در آزمون سوم ۷۵
- شکل ۶-۱۲: نمودار آزمون چهارم ۷۵
- شکل ۶-۱۳: بخشی از کد خروجی پیاده سازی شده مرحله اول آزمون چهارم ۷۶
- شکل ۶-۱۴: پیاده سازی از سمت کاربر برای متدهای تعریف شده در آزمون چهارم ۷۶
- شکل ۶-۱۵: کد استفاده شده در آزمون چهارم ۷۷

بخش اول: مقدمه

فصل اول

مقدمه

اهداف پروژه:

هدف پروژه ساخت یک نرم افزار است که به توسعه برنامه نویسی شیء گرا در زبان سی کمک کند. این نرم افزار به برنامه نویسان اجازه خواهد داد که ساختمان داده ای داشته باشند که قابلیت ارث بری و چند ریختی^۱ دارد و صاحب متد^۲ و سازنده و تخریب گر است. همچنین معماری نمودار کلاس را بررسی می کند که دارای اشکالاتی مانند داشتن حلقه در ارث بری و یا ممکن نبودن ساختن شیء مربوط به کلاس نباشد و همچنین به کاربر در صورت داشتن تکرار و نقص های ابتدایی در طراحی اخطار بدهد.

اجزاء پروژه:

نرم افزار در عمل دو نوع ورودی می تواند بگیرد. ورودی نوع اول اطلاعات کلاس هاست که در آن جزئیاتی مانند نام کلاس، نام کلاس پدر، سازنده ها و ... وجود دارد. این اطلاعات از دو طریق فایلی در قالب XML و یا رابط کاربری گرافیکی^۳ قابل دریافت می باشند.

این ورودی در دو بخش متفاوت که یکبار تکرار می شود بررسی شده است. در بخش اول اجزای داده برای جلوگیری از خطای کامپایل بررسی می شود. این بررسی درست بودن دستوری لغات و برابر نبودن^۱ سامی اجزای کلاس ها با همدیگر و همچنین موجودیت کلاس پدر^۴ را شامل می شود.

در بخش دوم وجود حلقه ارث بری و ممکن بودن ساخت اشیاء بررسی می شود. تکرار بررسی بعد از اعمال ارث بری انجام می شود.

^۱ Polymorphism

^۲ Method

^۳ Graphical user interface(GUI)

^۴ Super class

بخش اول: مقدمه

سپس بعد از بررسی و در صورت نداشتن خطا، فرایند ها، بخشی از اطلاعات کلاس ها و قالب های مخصوص پر کردن متد ها به عنوان خروجی داده می شوند.

ورودی نوع دوم شامل سرایندها، کد های کاربر و اطلاعات خروجی داده شده کلاس ها در بخش اول می باشد که کد های قابل اجرا در زبان سی را برمی گرداند.

همچنین برای بخش دوم برنامه که کد را به زبان سی انتقال می دهد تست نوشته است که می توان آن ها را اجرا کرد.

بخش دوم: مفاهیم پایه

فصل دوم

مفاهیم پایه

۲-۱ ویژگی های پروژه

۲-۱-۱ مهندسی نرم افزار به کمک کامپیوتر^۵

دامنه ای از ابزار ها و نرم افزارها برای طراحی و پیاده سازی نرم برنامه ها هستند.

این ابزار ها برای توسعه بدون نقص و قابل نگهداری استفاده می شوند و در طول چرخه عمر نرم افزار به مهندسان کمک می کنند.

نرم افزار های مهندسی نرم افزار به کمک کامپیوتر به سه گروه تقسیم می شوند:

۱- ابزاری هایی برای پشتیبانی از کار خاص در چرخه تولید نرم افزار.

۲- میز کار هایی که ترکیب چند ابزار هستند که با تمرکز بر روی یک بخش خاص از چرخه زندگی نرم افزار هستند.

۳- میز کار هایی که ترکیب چند ابزار هستند و در طول چرخه زندگی نرم افزار پشتیبانی کامل را انجام می دهند. [1]

یکی از این گروه نرم افزار ها Visual Paradigm که یک ابزار UML^۶ است می باشد.

۲-۱-۲ انتقال کد^۷

به طور معمول یک کد با در نظر گرفتن یک محیط محاسباتی خاص توسعه می یابد. بنابراین وقتی قرار است کدی که در یک محیط محاسباتی نوشته شده است در محیط متفاوتی اجرا شود، بخشی از کد باید اصلاح شود. به فعالیتی که کد را اصلاح می کند تا در یک محیط محاسباتی متفاوت قابل اجرا باشد انتقال کد می گویند. [2]

^۵ case

^۶ workbench

^۷ Unified Modeling Language

^۸ Code porting

۲-۱-۳ ترنسکامپایلر^۹

ترنسکامپایلر یا کامپایلر مبدا به مبدا ابزاری است که کد نوشته شده در یک زبان برنامه نویسی را به عنوان ورودی می گیرد و کد را به یک زبان دیگر بر می گرداند.

تفاوت ترنسکامپایلر با کامپایلر های عادی در سطح انتزاع است. کامپایلر یک کد با زبان سطح بالاتر مانند سی را به یک کد سطح پایین تر مانند کد ماشین تبدیل می کند ولی ترنسکامپایلر یک کد را به یک کد تقریباً هم سطح تبدیل می کند.

کامپایلر مبدا به مبدا چندین کاربرد دارد.

کاربرد اول آن در موازی سازی خود کار است که یک برنامه سطح بالا به طور مداوم می گیرد و آن را تبدیل می کند.

کاربرد دیگر آن تبدیل در کدهای موروثی^{۱۰} و یا رابط های برنامه نویسی نرم افزار^{۱۱} ناسازگار با نسخه های قدیمی است که آن را در نسخه بعدی زبان استفاده کنند. [3]

۲-۲ الگو های برنامه نویسی

۲-۲-۱ برنامه نویسی دستوری^{۱۲}

در یک جمله دستوری، فاعل جمله به صورت ضمنی است.

به طور مثال در جمله مقدار x را برابر ۵ قرار بده فاعل کسی است که به او دستور داده شده است و در یک برنامه، فاعل در واقع کامپیوتر می باشد.

زبان های برنامه نویسی دستوری با دنباله ای از پیوند ها و با حالات و دستوراتی که حالت را تغییر می دهند توصیف می شوند. برنامه ها دنباله های از پیوند ها (تغییر حالات) می باشند که در آن ها ممکن است یک نام در یک نقطه از برنامه به یک شی متصل بشود (مانند تعریف متغیر و قراردادن مقدار ابتدایی در آن) و بعداً به

^۹ transcompiler

^{۱۰} legacy code

^{۱۱} Application Programming Interface

^{۱۲} Imperative programming

یک شیء متفاوت متصل شود (مانند تغییر مقدار یک متغیر). همچنین ترتیب این پیوند ها بسیار مهم است و در نتیجه یکی از مهم ترین موضوعات الگو دستوری^{۱۳} کنترل کردن دنباله پیوند هاست.

مثال:

| | | | |
|---|-------------|---|-------------|
| 1 | $x = x * 2$ | 1 | $x = 7$ |
| 2 | $x = 7$ | 2 | $x = x * 2$ |

شکل ۲-۱: تغییر ترتیب ۲ خط

در حالت سمت راست مقدار داخل متغیر x برابر ۱۴ خواهد بود ولی در سمت چپ این مقدار ۷ است. همچنین در زمینه زبان های برنامه نویسی دستوری، یک حالت فراتر از ارتباط بین متغیر ها و مقادیر است و همچنین شامل محل کنترل در برنامه هم می باشد. الگوی برنامه نویسی دستوری خود انتزاعی از رایانه های واقعی است که بر اساس ماشین های تورینگ و ماشین وان نیومن^{۱۴} با رجیستر ها و حافظه آن هاست. [4] در مقابل الگوی برنامه نویسی دستوری، الگوی برنامه نویسی اعلامی^{۱۵} قرار دارد که برنامه نویسی تابعی زیر شاخه ای از آن است.

۲-۲-۲ برنامه نویسی ساخت یافته^{۱۶}

در سال ۱۹۶۹ دایکسترا^{۱۷} استدلال کرد که در توسعه یک برنامه باید در ابتدا وظایف اصلی که باید انجام بدهد را ترسیم کرد و سپس پی در پی این وظایف را به وظیفه های کوچکتر تبدیل کرد تا به سطحی برسیم که در آن هر وظیفه باقیمانده را بتوان با عملیات های پایه (مانند جمع و ضرب و مساوی) بیان کرد. این کار در انتها باعث می شود که وظایف هم بسیار کوچک هستند که بتوان آن ها را حل کرد و هم از همدیگر جدا و مستقل هستند که بتوان به صورت مستقل آن ها را حل کرد. مثالی که خود دایکسترا در این باره زده را در زیر می نویسم:

^{۱۳} Imperative paradigm

^{۱۴} Von Neumann

^{۱۵} Declarative programming paradigm

^{۱۶} Structured programming

^{۱۷} Edsger w. Dijkstra

فرض کنیم مسئله ما این است که ۱۰۰۰ عدد اول ابتدایی را چاپ کنیم. برنامه در ابتدا به شکل زیر است:

=====

شروع

۱۰۰۰ عدد اول ابتدایی را چاپ کن.

پایان

=====

حال ما می خواهیم برنامه را به دو زیر برنامه باز تعریف کنیم.

زیر برنامه اول را می توان پیدا کردن ۱۰۰۰ عدد اول تعریف کرد و زیر برنامه دوم را می توان چاپ ۱۰۰۰ عدد پیدا شده دانست.

=====

شروع

جدول p را تعریف کن.

جدول p را با ۱۰۰۰ عدد اول ابتدایی پر کن.

محتویات جدول p را چاپ کن.

=====

این تقسیم بندی باعث می شود که ما دو مسئله جداگانه که ساده تر و مستقل هستند را حل کنیم و می توانیم از دو برنامه نویس بخواهیم که این دو مسئله را حل کنند.

حال ما می خواهیم هر زیر برنامه را تقسیم کنیم تا در هر زیر برنامه جزئیات بیشتری مشخص باشد و برای برنامه نویس ساده تر باشد.

=====

شروع

آرایه ۱۰۰۰ تایی عددی p را تعریف کن.

یک حلقه برای متغیر k بساز به نحوی که k به ترتیب از ۱ تا ۱۰۰۰ باشد.

عدد خانه k ام آرایه p را برابر k امین عدد اول قرار بده.

یک حلقه برای متغیر k بساز به نحوی که k به ترتیب از ۱ تا ۱۰۰۰ باشد.

عدد خانه k ام آرایه p را چاپ کن.

=====

برنامه بالا برنامه ای به مراتب از نظر فهم ساده تر است. [5]

البته همانطور که نگاه می کنید این تقسیم بندی بهینه نیست. در اینجا ما شاهد تکرار شدن یک حلقه هستیم که دو بار تکرار شده است و می توان با یکبار استفاده از آن فهم مسئله را ساده تر کرد و همچنین به برنامه سرعت بخشید.

حال ما تقسیم بندی منطقی تری را ارائه می دهیم و همانطور که می بینید الگوریتم نوشته زیر بسیار مناسب تر است.

=====

شروع

آرایه ۱۰۰۰ تایی عددی p را تعریف کن.

یک حلقه برای متغیر k بساز به نحوی که k به ترتیب از ۱ تا ۱۰۰۰ باشد.

عدد خانه k ام آرایه p را برابر k امین عدد اول قرار بده.

عدد خانه k ام آرایه p را چاپ کن.

=====

۲-۲-۳ برنامه نویسی رویه ای^{۱۸}

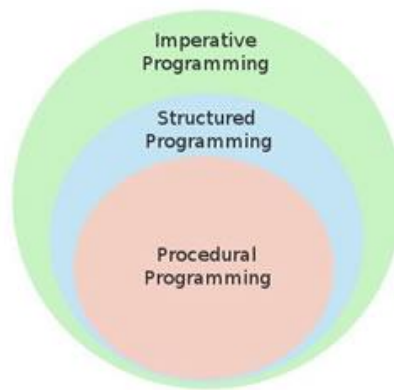
برنامه نویسی رویه ای مشتقی از برنامه نویسی ساخت یافته است که خود برنامه نویسی ساخت یافته مشتقی از برنامه نویسی دستوری می باشد. این الگوی برنامه نویسی بر اساس مفهوم فراخوانی روند^{۱۹} هاست. روند ها دارای یک سری گام های محاسباتی هستند.

برنامه نویسی رویه ای به ما کمک می کند تا بخش هایی از کد را بتوانیم دوباره و بدون کپی کردن استفاده کنیم و همچنین راحت تر بتوانیم جریان برنامه را پیگیری کنیم. زبان های متعددی هستند در برنامه نویسی رویه استفاده می شوند.

FORTRAN, ALGOL, COBOL, BASIC, C

همچنین زبان هایی هم هستند که این نوع برنامه نویسی را حمایت می کنند.

C++, Java, Python ...



شکل ۲-۲: ارتباط برنامه نویسی دستوری، ساخت یافته و رویه ای با همدیگر

^{۱۸} Procedural programming

^{۱۹} process

۲-۲-۴ برنامه نویسی شی گرا^{۲۰}

برنامه نویسی شی گرا در حال حاضر پر استفاده ترین الگوی برنامه نویسی در شرکت ها و صنعت است. این الگو می خواهد هویت داده به عنوان عناصر منفعل که رویه ها آن ها را تغییر می دهند را به داده ها به عنوان عناصر فعال که با محیط خود در تعامل هستند یا در واقع جریان کنترل را به اشیائی که در حال تعامل هستند تغییر بدهد.

برنامه نویسی شی گرا از برنامه های توسعه یافته از برنامه های شبیه سازی است. مدل مفهومی مورد استفاده این است که ساختار شبیه سازی باید محیطی را که شبیه سازی می کند را منعکس کند. به عنوان مثال، اگر قرار است یک فرایند صنعتی شبیه سازی شود، باید برای هر موجودیتی در فرایند یک شی وجود داشته باشد و اشیاء با ارسال پیام با یکدیگر ارتباط برقرار می کنند.

می توانیم نتیجه بگیریم که برنامه نویسی در یک زبان برنامه نویسی دستوری مانند C با برنامه نویسی در یک زبان برنامه نویسی شی گرا مانند جاوا متفاوت است. توسعه در یک زبان برنامه نویسی دستوری برنامه نویس را ملزم می کند که از نظر ساختار داده و الگوریتم هایی به تغییر ساختار داده فکر کند. یعنی داده ها در یک ساختار داده قرار می گیرد و ساختار داده با روش های مختلف تغییر داده می شوند.

از آن سمت برنامه نویسی در زبان های برنامه نویسی شی گرا، برنامه نویس را ملزم می کند تا بر اساس سلسله مراتب اشیاء و ویژگی های موجود در اشیاء فکر کند. [4]

زبان های برنامه نویسی که با اشیاء و کلاس ها هستند معمولاً دارای چهار ویژگی هست. حال ما چهار مفهوم اساسی در زبان های شی گرا وجود دارد که آن ها را شرح می دهیم.

۲-۲-۴-۱ جست و جو پویا^{۲۱}

جست و جو پویا بدین معناست که وقتی پیامی به یک شی ارسال می شود، متودی که باید اجرا بشود توسط روشی که شی پیاده سازی شده است مشخص می شود. می توان گفت این شی می باشد که انتخاب می کند

^{۲۰} Object Oriented programming

^{۲۱} Dynamic lookup

چطور پاسخ بدهد. در زبان های رویه ای اجزای ایستا^{۲۲} هستند که با استفاده از اشاره گر و یا مقداری که برای شیء تعیین شده است به این پیام ها پاسخ می دهند.

۲-۲-۴-۲ انتزاع^{۲۳}

انتزاع در برنامه نویسی شیء گرا بدین معناست که جزئیات پیاده سازی در داخل یک واحد برنامه با رابط کاربری خاص پنهان شده است. برای اشیاء رابط کاربری معمولاً گروهی از متد های عمومی هستند که داده های شیء را تغییر می دهند. در زبان های مدرن که شیء گرا هستند، دسترسی به اشیاء به عملگر های عمومی محدود شده است.

۲-۲-۴-۳ زیرنوع^{۲۴}

زیر نوع در برنامه نویسی به این معناست اگر یک شیء دارای تمام قابلیت های شیء دیگر باشد، ما می توانیم از آن شیء در هر زمینه ای که شیء دیگر را انتظار داریم استفاده کنیم. به طور دقیق تر می توان گفت که زیرنوع رابطه ای بر روی انواع است که اجازه می دهد از مقادیر یک نوع به جای مقادیر نوع دیگر استفاده شود. [5]

۲-۲-۴-۴ ارث بری^{۲۵}

ارث بری یک قابلیت زبان است که اجازه می دهد اشیاء جدید از اشیاء موجود تعریف شوند. همچنین می توان گفت ارث بری مکانیسمی است که در آن کلاس های جدید از کلاس های موجود که کلاس مافوق^{۲۶} مشتق گرفته می شوند. همچنین کلاس ها در بعضی از زبان ها می توانند از چند کلاس مشتق گرفته شده باشند.

^{۲۲} static

^{۲۳} Abstraction

^{۲۴} Subtyping

^{۲۵} Inheritance

^{۲۶} Super class

ارث بری شبیه زیرنوع است ولی تفاوت هایی دارد. زیرنوع باعث می شود که یک نوع داده شده جایگزین نوع یا انتزاع دیگری شود و این می تواند رابطه ای بین زیرنوع ها و برخی انتزاعات دیگر وابسته به زبان برنامه نویسی به صورت ضمنی یا صریح برقرار کند. این رابطه مانند کد زیر به زبان سی پلاس پلاس از طریق ارث بری به دست آمده است.

```
class A {
public:
    void DoSomethingALike() const {}
};

class B : public A {
public:
    void DoSomethingBLike() const {}
};

void UseAnA(const A& a) {
    a.DoSomethingALike();
}

void SomeFunc() {
    B b;
    UseAnA(b); // b can be substituted for an A.
}
```

شکل ۲-۳: استفاده از زیرنوع به وسیله ارث بری

در کد بالا یک رابطه صریح ارثی بین دو کلاس A و B ایجاد شده است. [6]

۲-۳ اصطلاحات مربوط به الگوهای برنامه نویسی

۲-۳-۱ مشخصه^{۲۷}

یک مشخصه ویژگی یک شی را تعریف می کند. مجموعه مقادیر همه مشخصه ها ویژگی های یک شی را تعریف می کنند که حالت شی هم به آن گفته می شود.

۲-۳-۲ متد^{۲۸}

یک متد در برنامه نویسی شی گرا یک رویه^{۲۹} است که با یک پیام و یک شی در ارتباط است. متد ها در واقع بخش رفتاری یک شی را بر عهده دارند و می توانند حالت آن را تغییر بدهند. همچنین متد های خاصی مانند سازنده و تخریب گر وجود دارند که به آن ها می پردازیم.

۲-۳-۳ سازنده^{۳۰}

یک سازنده متودی است که ابتدای عمر یک شی برای ایجاد و راه اندازی اولیه شی، فراخوانی می شود. سازنده ها ممکن است پارامتر داشته باشند ولی معمولاً مقادیر بازگشتی^{۳۱} ندارند.

۲-۳-۴ تخریب گر^{۳۲}

یک تخریب گر متودی است که انتهای عمر یک شی فراخوانی می شود و معمولاً وظیفه آزاد سازی متغیرها را داراست. همچنین کار های مورد نیاز قبل از آزاد سازی منابع را هم می تواند بر عهده بگیرد. تخریب گر ها معمولاً پارامتر و مقادیر بازگشتی ندارند.

^{۲۷} Attribute

^{۲۸} Method

^{۲۹} procedure

^{۳۰} Constructor

^{۳۱} Return Value

^{۳۲} Destructor

۲-۳-۵ چندریختی

چندریختی به سازه هایی اطلاق می شوند که در صورت نیاز می توانند انواع مختلفی را به خود بگیرند. به عنوان مثال، یک تابع که می تواند طول هر نوع لیست را محاسبه کند، چند شکل است. ۳ نوع چند ریختی در زبان های امروزی موجود است.

۱- چندریختی پارامتری:

که در آن یک تابع می تواند برای هر آرگومانی که انواع آن با نوع عبارت^{۳۳} مطابق است. در واقع چندریختی پارامتری به ما اجازه می دهد که یک کد را برای هر نوعی اجرا کنیم. برای مثال فرض کنید که ما تابع minimum با دو ورودی به شکل زیر داریم:

```
int minimum(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
```

شکل ۲-۴: تابع minimum با دو ورودی با نوع int

حال فرض کنید ما بخواهیم برای انواع دیگر این تابع را بنویسیم. آنگاه مجبوریم به ازای تمام انواع متفاوت یک تابع متفاوت بنویسیم. ولی با چندریختی پارامتری این مشکل حل می شود.

```
template <class Type>
Type minimum(Type a, Type b)
{
    if(a < b)
        return a;
    return b;
}
```

شکل ۲-۵: تابع minimum با استفاده از قابلیت template در زبان سی پلاس پلاس

^{۳۳} expression

۲- چندریختی زیرگونه^{۳۴}:

این توانایی استفاده از کلاس های مشتق شده از طریق اشاره گر ها و ارجاع^{۳۵} هاست. این توانایی از طریق متود غالب کردن^{۳۶} به دست می آید. این توانایی همچنین چندریختی زمان اجرا^{۳۷} هم می نامند. به طور مثال خانواده بیولوژیکی گربه سانان همه باید بتوانند صدای گربه (meow) را داشته باشند. حال ما کلاس پایه گربه سانان را Felid می نامیم و تمامی کلاس های گربه سان متد meow را غالب^{۳۸} کنند.

```
class Felid {
public:
    virtual void meow() = 0;
};

class Cat : public Felid {
public:
    void meow() { std::cout << "Meowing like a regular cat! meow!\n"; }
};

class Tiger : public Felid {
public:
    void meow() { std::cout << "Meowing like a tiger! MREOWWW!\n"; }
};

class Ocelot : public Felid {
public:
    void meow() { std::cout << "Meowing like an ocelot! mews!\n"; }
};
```

شکل ۲-۶: مثال چندریختی زیرگونه متد ها در کلاس به زبان C++

^{۳۴} Subtype Polymorphism

^{۳۵} Reference

^{۳۶} method overriding

^{۳۷} Runtime polymorphism

^{۳۸} override

```
void do_meowing(Felid *felid) {  
    felid->meow();  
}
```

```
int main() {  
    Cat cat;  
    Tiger tiger;  
    Ocelot ocelot;  
  
    do_meowing(&cat);  
    do_meowing(&tiger);  
    do_meowing(&ocelot);  
}
```

```
result:  
Meowing like a regular cat! meow!  
Meowing like a tiger! MREOWWW!  
Meowing like an ocelot! mews!
```

شکل ۲-۷: نتیجه مثال چندریختی زیرگونه متد ها در کلاس به زبان سی پلاس پلاس

۳- سربار گذاری^{۳۹}

سربار گذاری و یا چندریختی تک کاره^{۴۰} اجازه می دهد که چند تابع با یک اسم برای تایپ های متفاوت بتوانند رفتار متفاوتی را نشان بدهند.

به طور مثال تابع minimum که در چندریختی پارامتری نشان داده شد می توان برای انواع متفاوت نوشت. این قابلیت در زبان سی پلاس پلاس و جاوا و همچنین بسیاری دیگر از زبان های مدرن دیگر موجود است. [7]

^{۳۹} Overloading

^{۴۰} Ad-hoc Polymorphism

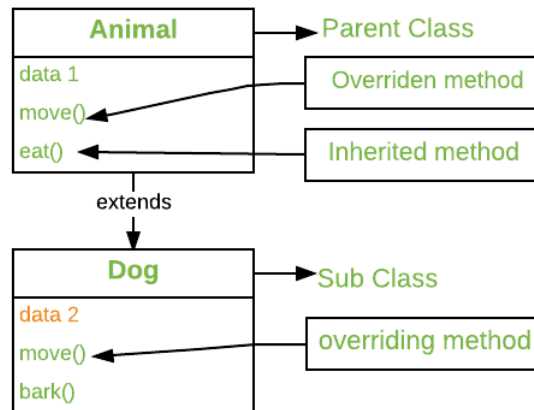
```
int add(int a, int b) {  
    return a + b;  
}  
  
std::string add(const char *a, const char *b) {  
    std::string result(a);  
    result += b;  
    return result;  
}  
  
int main() {  
    std::cout << add(5, 9) << std::endl;  
    std::cout << add("hello ", "world") << std::endl;  
}  
  
result:  
14  
hello world
```

شکل ۲-۸: مثال سربرارگذاری توابع به زبان سی پلاس پلاس

۲-۳-۶ متد غالب کردن

در هر زبان شیء گرا، غالب کردن قابلیت است که به کلاس های فرزند اجازه می دهد تا پیاده سازی خاصی از متودی که قبلا توسط یکی از کلاس های پدر پیاده سازی یا تعریف شده است را برای کلاس خود ارائه دهد. وقتی که سرایند یک متد (اسم متد، پارامترها، نوع خروج) در کلاس های فرزند با سرایند متد دیگر برابر است، متد کلاس فرزند غالب می شود.

این قابلیت در زبان های بسیاری مانند پایتون، جاوا، سی پلاس پلاس و ... وجود دارد و باعث شده از چندریختی زیرگونه ها در این زبان ها ممکن باشد. [8]



شکل ۲-۹: مثال ۲ کلاس که متد غالب شده دارند.

۲-۴ زبان سی

زبان سی یک زبان دستوری، ساخت یافته و رویه ایست. سی دارای کامپایلرهای متفاوت، پیاده سازیهای متفاوت است با این حال یکی از رایج ترین استانداردهای آن برای موسسه استاندارد های ملی آمریکا^{۴۱} می باشد. از این زبان در هسته سیستم عامل ها و بسیاری از برنامه های سخت افزاری استفاده می شود. در اینجا چند قابلیت و امکان مهم این زبان را بررسی می کنیم.

۲-۴-۱ سربار گذاری تابع^{۴۲}

زبان سی دارای این قابلیت نیست ولی این قابلیت در زبان سی پلاس پلاس به آن اضافه شده است. دو مثال زیر نشان می دهد تفاوت داشتن این قابلیت چه کمک بزرگی به توسعه دهنده است و همچنین در اینجا ماکروهای مخصوصی که در این پروژه به سربار گذاری متود و غالب کردن متود کمک کرده اند را معرفی می کنیم.

^{۴۱} ANSI

^{۴۲} Function Overloading

```
union x
{
    int x1,
        x2;
};

union y
{
    int x1;
    union x m;
};
```

شکل ۲-۱۰: تعریف ۲ ساختار X و Y

```
void print(union x* a) {
    printf("union x: %d\n", a->x1);
}

void print(union y* a) {
    printf("union y: %d\n", a->x1);
}

void print(union y* a, union x* b) {
    printf("union xy: %d\n", a->x1);
}

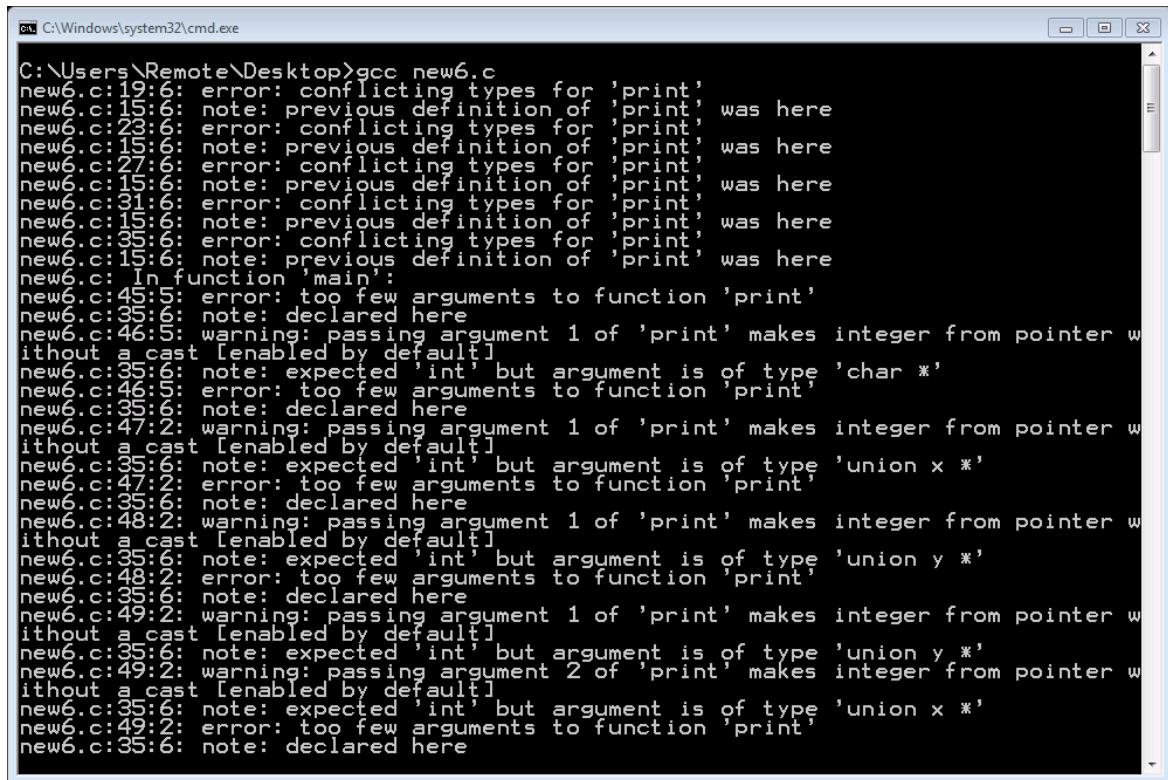
void print(int i, int j) {
    printf("int: %d %d\n", i, j);
}

void print(char* s) {
    printf("char*: %s\n", s);
}

void print(int i, int j, int v) {
    puts("triple");
}
```

شکل ۲-۱۱: تعریف چند تابع با اسم یکسان print در سی پلاس پلاس

در بالا در زبان سی پلاس پلاس چند تابع را با اسم یکسان تعریف کردیم و به درستی کامپایل شد. ولی در زبان سی با خطای زیر رو به رو هستیم:



```
C:\Windows\system32\cmd.exe
C:\Users\Remote\Desktop>gcc new6.c
new6.c:19:6: error: conflicting types for 'print'
new6.c:15:6: note: previous definition of 'print' was here
new6.c:23:6: error: conflicting types for 'print'
new6.c:15:6: note: previous definition of 'print' was here
new6.c:27:6: error: conflicting types for 'print'
new6.c:15:6: note: previous definition of 'print' was here
new6.c:31:6: error: conflicting types for 'print'
new6.c:15:6: note: previous definition of 'print' was here
new6.c:35:6: error: conflicting types for 'print'
new6.c:15:6: note: previous definition of 'print' was here
new6.c: In function 'main':
new6.c:45:5: error: too few arguments to function 'print'
new6.c:35:6: note: declared here
new6.c:46:5: warning: passing argument 1 of 'print' makes integer from pointer without a cast [enabled by default]
new6.c:35:6: note: expected 'int' but argument is of type 'char *'
new6.c:46:5: error: too few arguments to function 'print'
new6.c:35:6: note: declared here
new6.c:47:2: warning: passing argument 1 of 'print' makes integer from pointer without a cast [enabled by default]
new6.c:35:6: note: expected 'int' but argument is of type 'union x *'
new6.c:47:2: error: too few arguments to function 'print'
new6.c:35:6: note: declared here
new6.c:48:2: warning: passing argument 1 of 'print' makes integer from pointer without a cast [enabled by default]
new6.c:35:6: note: expected 'int' but argument is of type 'union y *'
new6.c:48:2: error: too few arguments to function 'print'
new6.c:35:6: note: declared here
new6.c:49:2: warning: passing argument 1 of 'print' makes integer from pointer without a cast [enabled by default]
new6.c:35:6: note: expected 'int' but argument is of type 'union y *'
new6.c:49:2: warning: passing argument 2 of 'print' makes integer from pointer without a cast [enabled by default]
new6.c:35:6: note: expected 'int' but argument is of type 'union x *'
new6.c:49:2: error: too few arguments to function 'print'
new6.c:35:6: note: declared here
```

شکل ۲-۱۲: نتیجه تعریف چند با اسم یکسان در سی

در بالا کامپایلر به ما خطای conflicting type را می دهد.

حال اگر ما بخواهیم برای زبان سی از یک اسم برای فراخوانی چند تابع استفاده کنیم راه های متفاوتی وجود دارد.

راه حل اول استفاده از CLANG و یا استفاده از سی ورژن ۱۱ می باشد که در این پروژه به این راه نمی پردازیم زیرا هدف ما از ابتدا عدم تغییر محیط کامپایل بوده است.

راه حل دیگر تعریف چند تابع و استفاده از پیش پردازنده^{۴۳} های زبان سی برای استفاده از یک نام برای آن هاست.

ابتدا چند تابع را جداگانه تعریف می کنیم.

```
void print_x(union x* a) {
    printf("union x: %d\n", a->x1);
}

void print_y(union y* a) {
    printf("union y: %d\n", a->x1);
}

void print_xy(union y* a, union x* b) {
    printf("union xy: %d\n", a->x1);
}

void print_int(int i, int j) {
    printf("int: %d %d\n", i, j);
}

void print_string(char* s) {
    printf("char*: %s\n", s);
}

void print_triple(int i, int j, int v)
{
    puts("triple");
}
```

شکل ۲-۱۳: تعریف چند تابع با اسم و امضای متفاوت در سی

حال ما می خواهیم با استفاده از پیش پردازنده ها کاری کنیم که با فراخوانی تابع print تمام این توابع بتوانند صدا زده شوند.

^{۴۳} preprocessor


```
#define CHOOSE __builtin_choose_expr
#define IFTYPE(X, T) __builtin_types_compatible_p(typeof(X), T)

#define print1(...) \
    CHOOSE(IFTYPE(FIRST(__VA_ARGS__), int), print_int, \
    CHOOSE(IFTYPE(FIRST(__VA_ARGS__), union x*), print_x, \
    CHOOSE(IFTYPE(FIRST(__VA_ARGS__), union y*), print_y, \
    print_string))) \
    (__VA_ARGS__)

#define print2(...) \
    CHOOSE(IFTYPE(FIRST(__VA_ARGS__), union y*) && \
    IFTYPE(SECOND(__VA_ARGS__), union x*) , print_xy, \
    print_int) \
    (__VA_ARGS__)

#define print3(...) \
    print_triple(__VA_ARGS__)

#define FIRST(A, ...) A
#define SECOND(A, B, ...) B

#define SELECT_N(X, _1, _2, _3, N, ...) N

#define print(...) SELECT_N(X, ##__VA_ARGS__, \
    print3, print2, print1, PUT0) \
    (__VA_ARGS__)
```

شکل ۲-۱۴: توانایی فراخوانی چند تابع با اسم `print` با استفاده از پیش پردازنده ها

برای فهم مثال در ادامه به پیش پردازنده ها در زبان سی می پردازیم.

سپس سه تابع توکار^{۴۴} `__builtin_choose_expr` و `__builtin_types_compatible_p` و `typeof` و را معرفی خواهیم کرد.

همچنین در شکل زیر تابع `main` را مشاهده می کنید که در هر دو کامپایلر خروجی یکسان می دهد.

^{۴۴} Built-in Function

```
int main(int argc, char* argv[]) {
    union x w;
    union y m;
    w.x1 = 70;
    m.m = w;
    print(1, 2);
    print("this");
    print(&w);
    print(&m);
    print(&m, &w);
    print(1,2,3);
    return 0;
}
```

شکل ۲-۱۵: استفاده از کلیدواژه print در تابع main

۲-۴-۲ پیش پردازنده

رایج ترین پیش پردازنده، پیش پردازنده C می باشد، که به صورت فراگیری در سی و فرزند آن سی پلاس پلاس، استفاده می شود. از این پیش پردازنده به منظور استفاده از خدمات معمول پیش پردازنده ای استفاده می شود. این پیش پردازنده قادر به انجام اعمال اولیه ای از قبیل همگردانی شرطی، جا دادن فایل در کد منبع، تعیین پیغام خطاهای زمان همگردانی و اعمال قواعد مخصوص ماشین مقصد به بخش های کد نهائی می باشد.[9]

۲-۴-۳ تابع توکار typeof

این تابع به ما کمک می کند که ارجاعی به نوع متغیر را داشته باشیم. این تابع کاربرد های زیادی دارد. یکی از این کاربرد ها تعریف یک ماکرو حداکثر است که هم امن است و هم از استفاده از پشته^{۴۵} برنامه جلوگیری می کند.

```
#define max(a,b) \
({ typedef (a) _a = (a); \
    typedef (b) _b = (b); \
    _a > _b ? _a : _b; })
```

^{۴۵} stack

در مثال بالا اگر a یک عدد باشد مقدار `typeof (a)` برابر `int` می باشد. [10]

۲-۴-۴ تابع `__builtin_types_compatible_p`

این تابع به ما کمک می کند تا متوجه شویم آیا دو تایپ با هم برابر هستند. همچنین این تابع توصیف کننده های سطح بالا^{۴۶} را نادیده می گیرد.

```
#define foo(x) \
({ \
    typeof (x) tmp = (x); \
    if (__builtin_types_compatible_p (typeof (x), long double)) \
        tmp = foo_long_double (tmp); \
    else if (__builtin_types_compatible_p (typeof (x), double)) \
        tmp = foo_double (tmp); \
    else if (__builtin_types_compatible_p (typeof (x), float)) \
        tmp = foo_float (tmp); \
    else \
        abort (); \
    tmp; \
})
```

شکل ۲-۱۶: استفاده از تابع توکار `__builtin_types_compatible_p`

در مثال بالا در صورت فراخوانی `foo`، وابسته به نوع متغیر x مقدار تابع مورد مطلوب را در `tmp` می ریزد.

۲-۴-۵ تابع توکار `__builtin_choose_expr`

این تابع با توجه در صورت `true` بودن مقدار ثابت پارامتر اول، پارامتر دوم را خروجی می دهد و در غیر این

صورت مقدار پارامتر سوم را خروجی می دهد. [11]

همانطور که در شکل ۲-۱۳ مشاهده می کنید، با استفاده از این تابع توانسته ایم در صورت فراخوانی `print`،

وابسته به تعداد پارامتر و نوع آن ها تابع مورد نظر را خروجی می دهد.

عملا با استفاده از سه تابع در پیش پردازنده می توان سربار گذاری تابع را به طور مصنوعی ایجاد کرد.

در فصل های آینده کاربرد این کار را شرح خواهم داد.

^{۴۶}Top level qualifiers

۲-۴-۶ اشاره گر

یک اشاره گر دارای یک آدرس از حافظه است که از مقدار nil شروع می شود. مقدار nil یک آدرس معتبر و برای این استفاده می شود که نشان بدهد اشاره گر در حالت حاضر نمی تواند به یک سلول حافظه ارجاع بدهد.

اشاره گر ها برای دو استفاده ساخته شده اند. اول اینکه اجازه آدرس دهی غیر مستقیم را بدهد و همچنین راهی را برای مدیریت حافظه هرم^{۴۷} فراهم آورده اند.

۲-۴-۷ union

یک union می تواند چند نوع ارائه از یک آدرس حافظه داشته باشد. همچنین از نظر ساختاری یک struct با یک union شبیه است.

اندازه یک union برابر اندازه بزرگترین عضو آن از نظر اندازه است.

قابل یک union به شکل زیر است:

```
union < name >
{
    < datatype > < 1st variable name >;
    < datatype > < 2nd variable name >;
    .
    .
    .
    < datatype > < nth variable name >;
} < union variable name >;
```

شکل ۲-۱۷: قالب تعریف union

^{۴۷} Heap memory

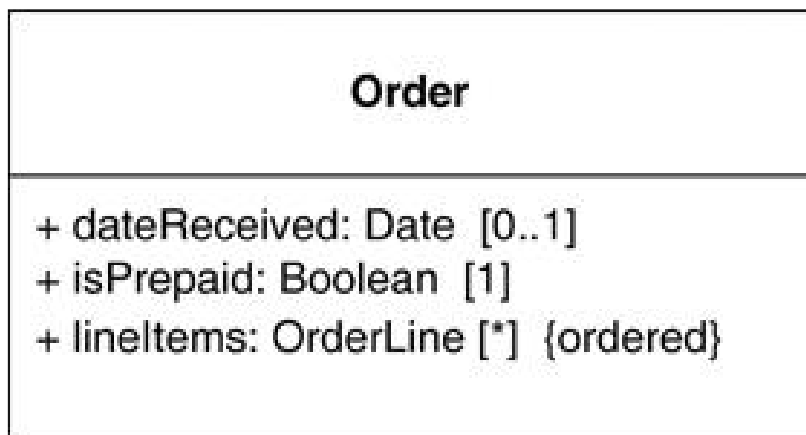
۲-۴-۸ زبان سی پلاس پلاس

زبان C++ یک چند الگو است که شی گرا و دستوری است.

این زبان توسط بیارنه استراس تروپ^{۴۸} پایه گذاری شد و در ابتدا سی با کلاس^{۴۹} نام داشت و هدف زبان این بود که قابلیت های سی مانند دستکاری در حافظه را داشته باشد و مانند زبان سیمولا شی گرا باشد.

۲-۵ نمودار کلاس^{۵۰}

یک نمودار کلاس نوع اشیاء و انواع روابط ایستایین آن ها را نشان می دهد. نمودار کلاس، اعضا^{۵۱} و عملگرهای یک کلاس و محدودیت هایی که در نحوه اتصال اشیاء اعمال می شود را نشان می دهد. اعضا یک کلاس ویژگی های ساختاری آن را نشان می دهد. خواص به دو گروه مشخصه ها و انجمن^{۵۲}ها تقسیم می شود. نماد مشخصه در داخل جعبه خود کلاس توصیف می کند.



شکل ۲-۱۸: تعریف مشخصه ها در داخل کلاس

هر مشخصه دارای اسم می باشد و نوع پدیداری^{۵۳}، نوع، مقدار اولیه و تعدد را می توان تعیین کرد.

^{۴۸} Bjarne Stroustrup

^{۴۹} C with classes

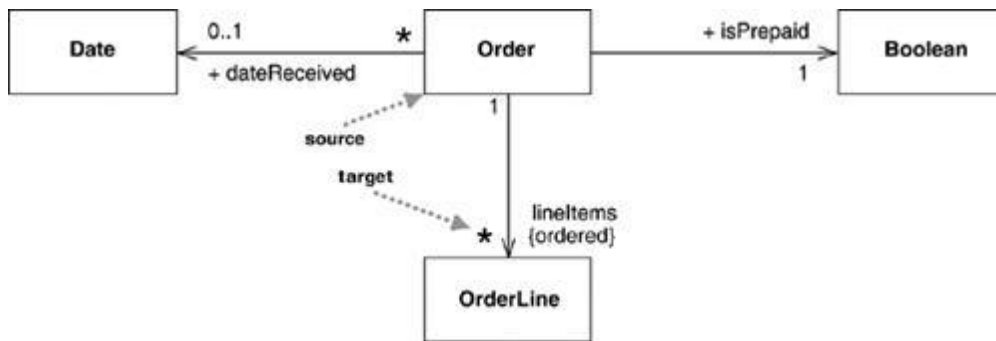
^{۵۰} Class Diagram

^{۵۱} Properties

^{۵۲} Association

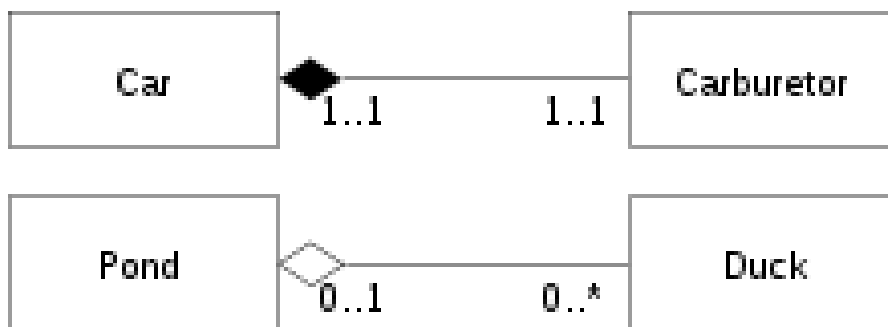
^{۵۳} visibility

یک گروه از اعضا انجمن ها می باشند. یک انجمن یک خط جهت دار بین دو کلاس است و نام و تعداد تکرار آن در انتهای خط است. شکل زیر معادل شکل ۱۸-۲ می باشد.



شکل ۱۹-۲: تعریف انجمن ها برای مشخصه

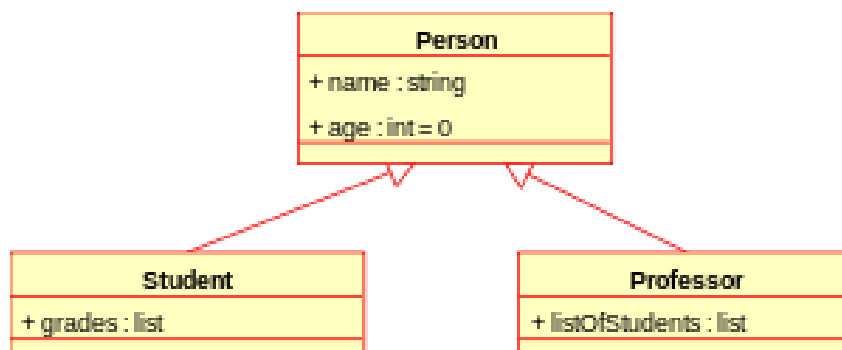
درباره تعداد اعضا حالت های متفاوتی را دارا هستیم. تعداد آن ها می تواند متفاوت باشد. می تواند حضور آن اختیاری باشد (به این معنی که دارای ۰ عضو یا هر تعدادی باشد). می تواند مشخص باشد (مثلا دقیقا ۱ یا دقیقا ۳۰). و یا اجازه بدهد در یک بازه خاص باشد. (از ۰ تا ۱ به طور یا حداقل ۱)



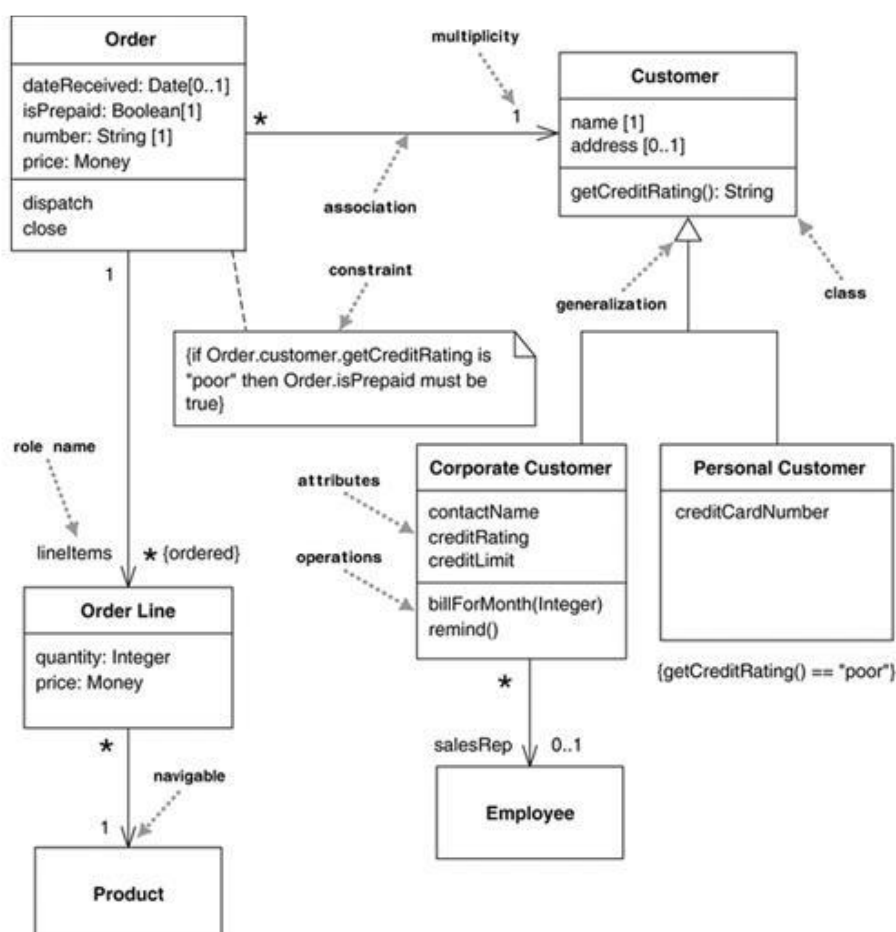
شکل ۲۰-۲: تعیین تعداد در نمودار کلاس

یکی از مهمترین ویژگی های یک کلاس داشتن متد هست که در نمودار کلاس پوشش داده شده اند. یک عملگر ها دارای اسم ، نوع پدیداری ، نوع خروجی ، لیست پارامتر های ورودی می باشد.

یکی از مهمترین ویژگی برنامه نویسی شی گرا وراثت است. در نمودار کلاس یک کلاس می تواند تعمیمی از کلاس پدرش باشد. [12]



شکل ۲-۲۱: تعیین وراثت در نمودار کلاس



شکل ۲-۲۲: یک نمودار کلاس کامل

۲-۶ تحلیل گر لغوی^{۵۴}

تحلیل گر لغوی معمولاً یک زیر برنامه است که کار اصلی آن خواندن حروف ورودی و تولید یک توالی از نشانه هاست. هنگامی که تحلیل گر نحوی^{۵۵} تحلیل گر لغوی را فراخوانی می کند، تحلیل گر تا زمانی که نشانه^{۵۶} بعدی مشخص شود، حروف ورودی را می خواند و نشانه ایجاد شده را به تحلیل گر نحوی برمی گرداند.

به طور کلی برای ساخت یک تحلیل گر لغوی دو راه را در پیش داریم، یک راه این هست که یک تحلیل گر لغوی با استفاده از ترسیم نمودارهای انتقال^{۵۷} بسازیم. راه دیگر استفاده از برنامه لکس در یکی از زبان های موجود است که ما این راه را بر می گزینیم. [13]

^{۵۴} Lexical Analyzer

^{۵۵} Syntax Analyzer

^{۵۶} Token

^{۵۷} Transition Diagrams

بخش سوم: کار های پیشین

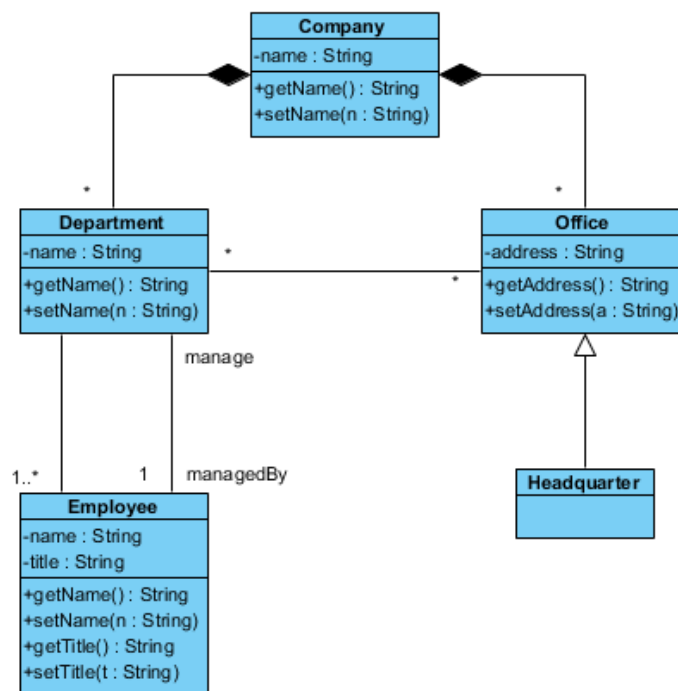
فصل سوم

کار های پیشین

تبدیل نمودار به کد سی و برنامه نویسی شی گرا در زبان سی موضوعات نامعمولی هستند و دلیلی آن نهفته در ماهیت زبان سی می باشد. زبان سی به خودی خود یک زبان شی گرا نمی باشد و همچنین زبان سی پلاس پلاس یکی زبان چند الگویی است که یکی از این الگو ها شی گرای می باشد و قابلیت های زبان سی را هم داراست. در حال حاضر برنامه ای در سطح اینترنت که قابلیت خواسته شده در این پروژه را پوشش داده باشد وجود ندارد ولی سه گروه کار انجام شده که مربوط به این پروژه هستند را در اینجا شرح داده می شود که به این پروژه مربوط می باشد.

۱-۳ تبدیل نمودار کلاس به کد

تعداد نرم افزار های تبدیل نمودار کلاس به کد زبان های شی گرا مانند جاوا و سی پلاس پلاس و معکوس کننده آن (یعنی تبدیل کد به نمودار) بسیار زیادند و همچنین کیفیت بالایی دارند.



شکل ۱-۳: یک نمودار کلاس ارائه شده در نرم افزار visual paradigm

شکل بالا یک نمودار کلاس است که به صورت نمونه در نرم افزار visual paradigm موجود است. خروجی زیر برای کلاس Employee می باشد.

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

class Employee {

private:
    Department manage;
    String name;
    String title;

public:
    String getName();

    void setName(String n);

    String getTitle();

    void setTitle(String t);
};

#endif

#include "Employee.h"

String Employee::getName() {
    return this->name;
}

void Employee::setName(String n) {
    this->name = n;
}

String Employee::getTitle() {
    return this->title;
}

void Employee::setTitle(String t) {
    this->title = t;
}
```

شکل ۲-۳: خروجی کد نمودار کلاس ارائه شده در نرم افزار visual paradigm

ابزار و توصیه‌ها برای تبدیل UML به زبان سی هم موجود است. هرچند کاربر خود باید با توجه به محدودیت‌های زبان سی متد‌ها را تعریف کند و در زبان از آن استفاده کند.

در یکی از اوراق سفید^{۵۸} منتشر شده در IBM برای استفاده از UML در برنامه‌های بی‌درنگ^{۵۹} به زبان برای نگاشت ویژگی‌های شی‌گرا چند توصیه کرده است. [14]

۱- کلاس‌ها تبدیل به struct بشوند.

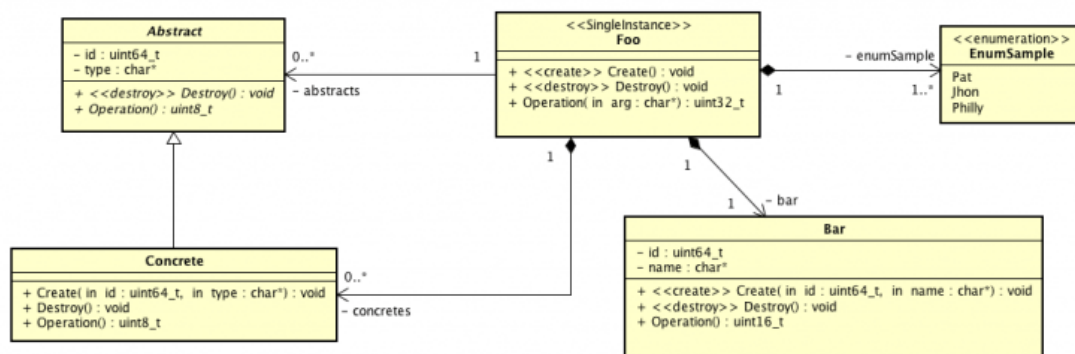
۲- داده‌ها مانند قبل بمانند و متد‌ها را با اشاره‌گرهایی به توابع جایگزین کنیم.

۳- تعمیم را با استفاده از تو در تویی struct‌هایی که معرف کلاس پایه هستند انجام بدهیم.

۴- چندریختی عملگرها را با استفاده از دستکاری نام^{۶۰} انجام دهیم. (که ما از این راهکار استفاده کرده‌ایم.)

ابزارهای مربوط به تبدیل UML به عنوان یکی از افزونه‌ها در نرم‌افزارهای توسعه نرم‌افزار مانند Eclipse، Atash و Sparx system موجود هستند.

ما مثالی از افزونه موجود در Atash با نام “UML2C Export” می‌زنیم.



شکل ۳-۳: دیاگرام نمونه در نرم‌افزار Atash

^{۵۸} White paper

^{۵۹} Real-time application

^{۶۰} Name-mangling

در شکل بالا کلاس Bar را داریم که نرم افزار برای این کلاس سه فایل تولید می کند. [15]

```
#ifndef _BAR_PRIVATE_H_
#define _BAR_PRIVATE_H_
#ifdef __cplusplus
extern "C" { /* } */
#endif

struct BarStruct {
uint64_t id;
char *name;
};

#ifdef __cplusplus
}
#endif
#endif /* _BAR_PRIVATE_H_ */
```

شکل ۳-۴: فایل تعریف ساختار کلاس Bar

```
#ifndef _BAR_H_
#define _BAR_H_
#ifdef __cplusplus
extern "C" { /* } */
#endif

typedef struct BarStruct *Bar;

void Bar_Create(Bar self, uint64_t id, const char *name);
void Bar_Destroy(Bar self);
uint16_t Bar_Operation(Bar self);

#include "BarPrivate.h"

#ifdef __cplusplus
}
#endif
#endif /* _BAR_H_ */
```

شکل ۳-۵: فایل سرایند متد های کلاس Bar

در تعریف سرایندهای متد همانطور که می‌توانید ببینید در واقع یک تابع را داریم که پارامتر ابتدایی آن اشاره‌گر به یک کلاس آن متد دارد. ما خود در پروژه این کار را کرده‌ایم.

```
#include "Bar.h"

void Bar_Create(Bar self, uint64_t id, const char *name)
{
    /* TODO */
}

void Bar_Destroy(Bar self)
{
    /* TODO */
}

uint16_t Bar_Operation(Bar self)
{
    /* TODO */
}
```

شکل ۳-۶: فایل تعریف متدهای کلاس Bar

۳-۲ کامپایلر سی پلاس پلاس به سی

قبل از ظهور ++g به عنوان کامپایلر، کامپایلر cfront کامپایلر اصلی مورد استفاده برای زبان سی پلاس پلاس بود.

روش کار این کامپایلر، تبدیل کد سی پلاس پلاس به کد سی بود. پیش پردازنده زبان را متوجه نمی‌شد و بیشتر کد توسط ترنسکامپایلر تولید می‌شد.

این کامپایلر مشکلات زیادی مانند عدم حمایت از رسیدگی به استثنا^{۶۱} را دارا بود و در نتیجه در سال ۱۹۹۳ رها شد. [16]

بعد ها کامپایلر Comeau C/C++ با رویکرد مشابه به بازار آمد و در حال حاضر llvm تبدیل کد زبان سی پلاس پلاس به کد سی را حمایت می‌کند.

با این وجود این نرم افزارها نیاز به حضور در محیط مورد نظر برای تولید کد هستند.

^{۶۱} Exception handling

۳-۳ برنامه نویسی شیء گرا در زبان سی

کتاب ها و جزوات متعددی در این زمینه منتشر شده است. یکی از این مثال ها کتاب برنامه نویسی شیء گرا در زبان سی به نوشته "Axel-Tobias Schreiner" می باشد. در این کتاب نویسنده روش های مخفی کردن اطلاعات، ارث بری، ساخت متد برای کلاس و ... آموزش داده شده است.

همچنین در هسته لینوکس تکنیک هایی که مربوط به برنامه نویسی شیء گرا استفاده شده است. به طور مثال ما در بخش هایی از کد روانه کردن متد^{۶۲} [17] و یا مخفی کردن اطلاعات [18] را شاهد هستیم.

^{۶۲} Method dispatch

بخش چهارم: دلایل برنامه نویسی شیء گرا در زبان سی

فصل چهارم

دلایل برنامه نویسی شیء گرا در زبان سی

در این بخش می خواهیم دلیل استفاده از الگوهای برنامه نویسی شیء گرا در زبان سی را بررسی کنیم.
برای بیان دلیل باید دو سوال را پاسخ بدهیم.

اولین سوال چرایی استفاده از الگوی شیء گرایی است.

دومین سوال دلیل استفاده از زبان سی با وجود زبان قدرتمندی مانند سی پلاس پلاس که قابلیت های زبان سی را داراست و از الگوی برنامه نویسی شیء گرا حمایت می کند.

۴-۱ دلایل استفاده از الگوی برنامه نویسی شیء گراء

برنامه نویسی شیء گرا تبدیل به بخش اساسی از توسعه نرم افزار شده است.
حال می خواهیم دلایل استفاده از این الگو را در اینجا شرح دهیم.

۴-۱-۱ مدولار بودن^{۶۳} برای عیب یابی آسان تر

یکی از فواید مخفی سازی این است که تمام اشیاء محدود هستند و عملکرد مربوط به خود را دارا هستند.
مدولار بودن اجازه می دهد در عیب یابی چند نفر همزمان بر روی چند بخش متفاوت کار کنند و احتمال تکراری بودن را پایین می آورد.

۴-۱-۲ قابلیت استفاده مجدد از کد

ارث بری سبب می شود که ما برای هر ساختار داده دوباره کد نویسی نکنیم و برای بخش های مشابه یک گروه فقط یکبار کد نویسی کنیم.

۴-۱-۳ انعطاف پذیری

یکی از فواید چندریختی انعطاف پذیری است که باعث می شود یک رفتار همگانی با اشیاء یک گروه داشته باشیم.

^{۶۳} modularity

۴-۱-۴ امنیت

انتزاع و مخفی سازی داده باعث می شود داده های بسیاری فیلتر شوند و فقط داده های مورد نیاز در دسترس برنامه نویس می باشد.

۴-۲ دلایل استفاده از زبان سی

شرایطی وجود دارد که استفاده از زبان سی اجباری است و ما این شرایط را توضیح می دهیم.

۴-۲-۱ ممکن نبودن استفاده از کامپایلر

شرایطی وجود دارد که ما از نظر سخت افزاری و یا نرم افزاری نمی توانیم از زبان سی استفاده کنیم. به طور مثال توسعه هسته سیستم عامل ها به زبانی غیر از سی بسیار سخت است و یا ریزکنترل گر^{۶۴} ها به زبان سی ورودی می گیرند.

۴-۲-۲ ادامه توسعه بر روی کد به زبان سی

بسیاری از برنامه ها و پروژه ها مانند هسته سیستم عامل لینوکس و گیت^{۶۵} که بسیار بزرگ هستند و به زبان سی نوشته شده اند. تغییر آن ها به نحوی که به زبان سی پلاس پلاس تبدیل شوند پرهزینه و بی فایده است.

^{۶۴} Microcontroller

^{۶۵} git

طراحی و پیاده سازی

۵-۱ قابلیت های ارائه شده در نرم افزار

نرم افزار پیاده سازی شده به برنامه نویسی شی گرا در زبان سی کمک می کند. در این قسمت می خواهیم قابلیت های ارائه شده را معرفی کنیم.

۵-۱-۱ متد

ساختارهای داده مانند کلاس ها می توانند متد داشته باشند و متدها به اعضای کلاس و متد های دیگر دسترسی داشته باشند.

۵-۱-۲ سازنده

ساختارهای داده مانند کلاس ها می توانند سازنده داشته باشند که مانند زبان سی پلاس پلاس در زمان تعریف و یا با استفاده از عملگر new کلاس را بسازند.

۵-۱-۳ تخریب گر

ساختارهای داده مانند کلاس ها می توانند تخریب گر داشته باشند که مانند زبان سی پلاس پلاس در زمان خروج از محدوده^{۶۶} متد خروج را صدا کند و یا با استفاده از عملگر delete کلاس را تخریب و حافظه آن را آزاد کند.

۵-۱-۴ ارث بری

کلاس ها می توانند تعمیم کلاس دیگری باشند. کلاس های فرزند مشخصه ها و متد ها به جز تخریب گر و سازنده پدر را دارند.

^{۶۶} scope

۵-۱-۵ سربار گذاری متد ها

متد های یک کلاس می توانند چند تابع با اسم یکسان و سازنده با پارامتر های ورودی متفاوت داشته باشند.

۵-۱-۶ متد غالب کردن

ساختارهای داده مانند کلاس ها می توانند سازنده داشته باشند که مانند زبان سی پلاس پلاس در زمان تعریف و یا با استفاده از عملگر new کلاس را بسازند.

۵-۱-۷ تبدیل نوع به بالا^{۶۷}

می توان اشیاء یک کلاس را به نوع بالا تبدیل و از آن ها استفاده کرد. باید توجه کرد که در پیاده سازی ما متد های پیاده سازی شده تا سطح کلاس تبدیل شده صدا زده می شوند و متد غالب شده صدا زده نمی شود.

۵-۱-۸ چک کردن خطاهای پایه

خطاهای پایه خطاهایی هستند که در باعث خطای کامپایل می شوند. در کد ۱۵ نوع خطای ممکن که در ورودی ها ممکن است به وجود بیاید چک می شود و در صورت پیدا شدن خطا برنامه اجازه ادامه کار را نمی دهد.

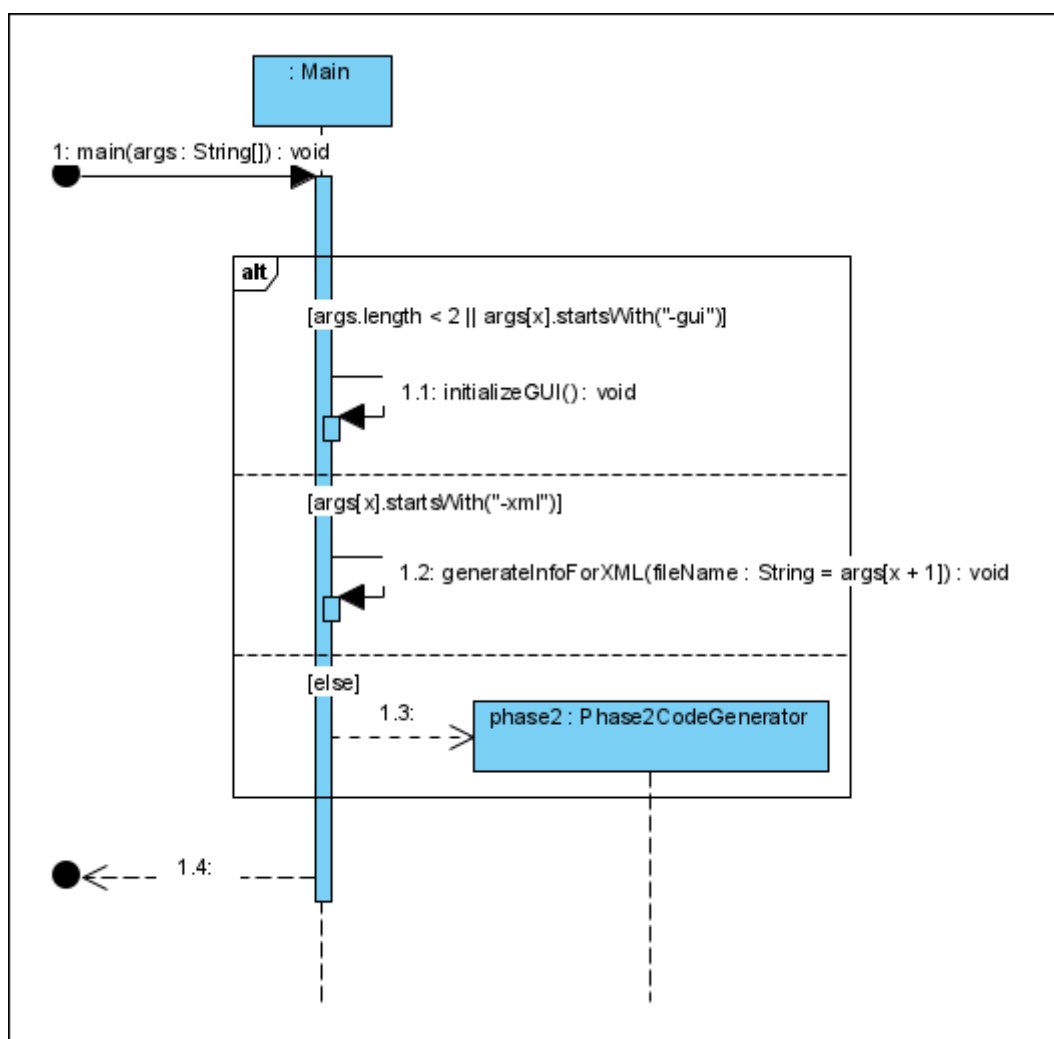
۵-۱-۹ بررسی دور های وابستگی در نمودار

وابستگی های بین دو کلاس حالت های متفاوتی می تواند داشته باشد. ۱۲ وابستگی متفاوت بین دو کلاس در نظر گرفته شده است. در صورتی که ساخت اشیاء از یک کلاس غیر ممکن باشد برنامه اجازه ادامه کار را نمی دهد و گرنه بعد از دادن خروجی نتیجه بررسی، کار را ادامه می دهد.

۲-۵ ورودی نرم افزار و کارکرد آن

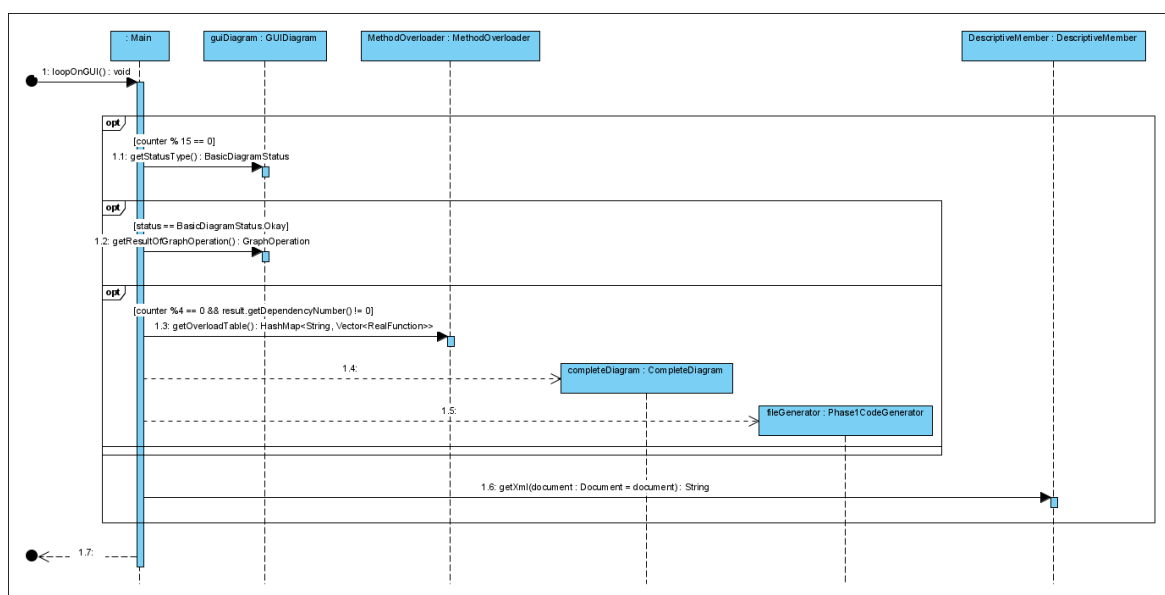
در ابتدا نرم افزار از ورودی اجرای برنامه را می گیرد و یکی از سه کار زیر را انجام می دهد.

- ۱- ورودی XML می گیرد و کدهای فاز یک را خروجی می دهد.
- ۲- ورودی گرافیکی می گیرد و XML و کدهای فاز یک را خروجی می دهد.
- ۳- کدهای کاربر و کدهای تغییر یافته فاز یک را می گیرد و کد به زبان سی تحویل می دهد.



شکل ۵-۱: نمودار گرفتن ورودی

بخش پنجم: طراحی و پیاده سازی



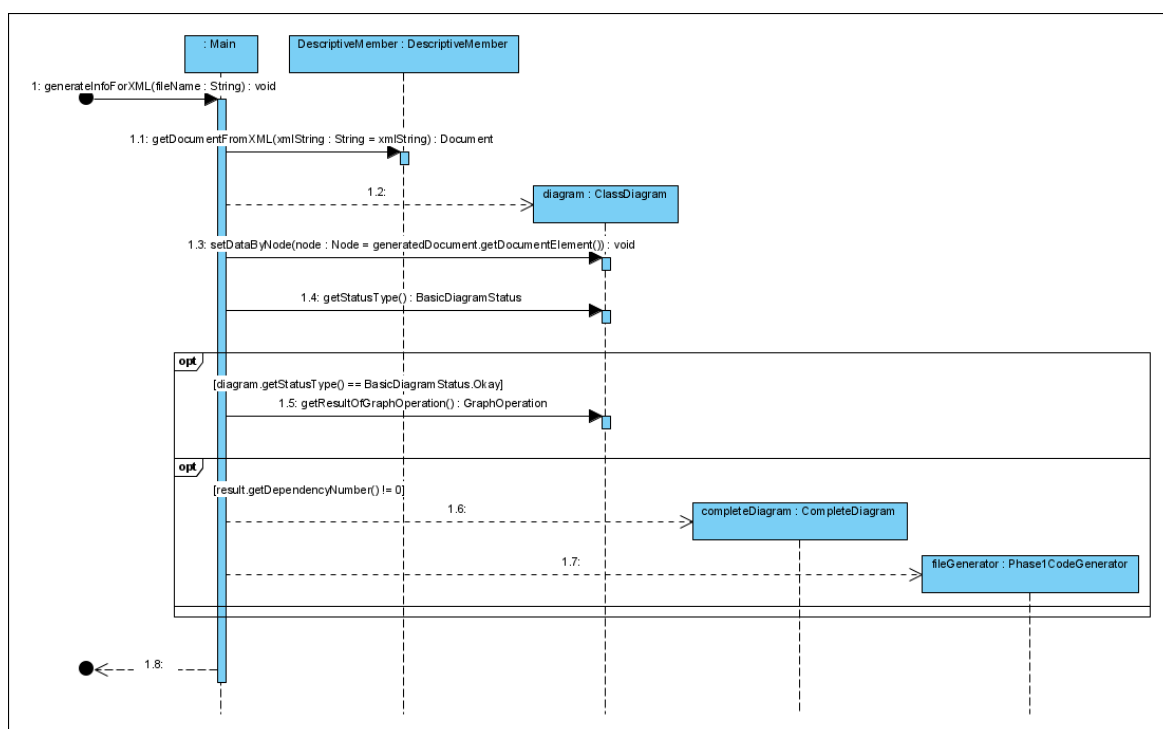
شکل ۵-۲: نمودار حلقه خروجی در حالت ورودی گرافیکی

در حالت گرافیکی بعد از ساخته شدن در یک حلقه بی نهایت ورودی کاربر بررسی می شود.

این حلقه کارهای زیر انجام می شود:

۱. هر ۳۰ ثانیه خطاهای پایه و وضعیت وابستگی کلاس ها اعلام می شود.
۲. در هر ۱۲۰ ثانیه در صورتی که خطای پایه وجود نداشته باشد و وضعیت وابستگی ۰ نباشد آنگاه:

۱. جدول بارگذاری متدها را پاک می کند.
۲. سپس نمودار کامل برای اعمال اثربری و اضافه کردن کد برای متمایز شدن توابع ساخته می شود.
۳. بعد از بررسی دوباره خطاهای وابستگی و پایه کد فاز یک تولید می شود.
۳. سپس فایل XML نمودار جهت استفاده دوباره کاربر خروجی داده می شود.



شکل ۳-۵: نمودار حلقه خروجی در حالت ورودی XML

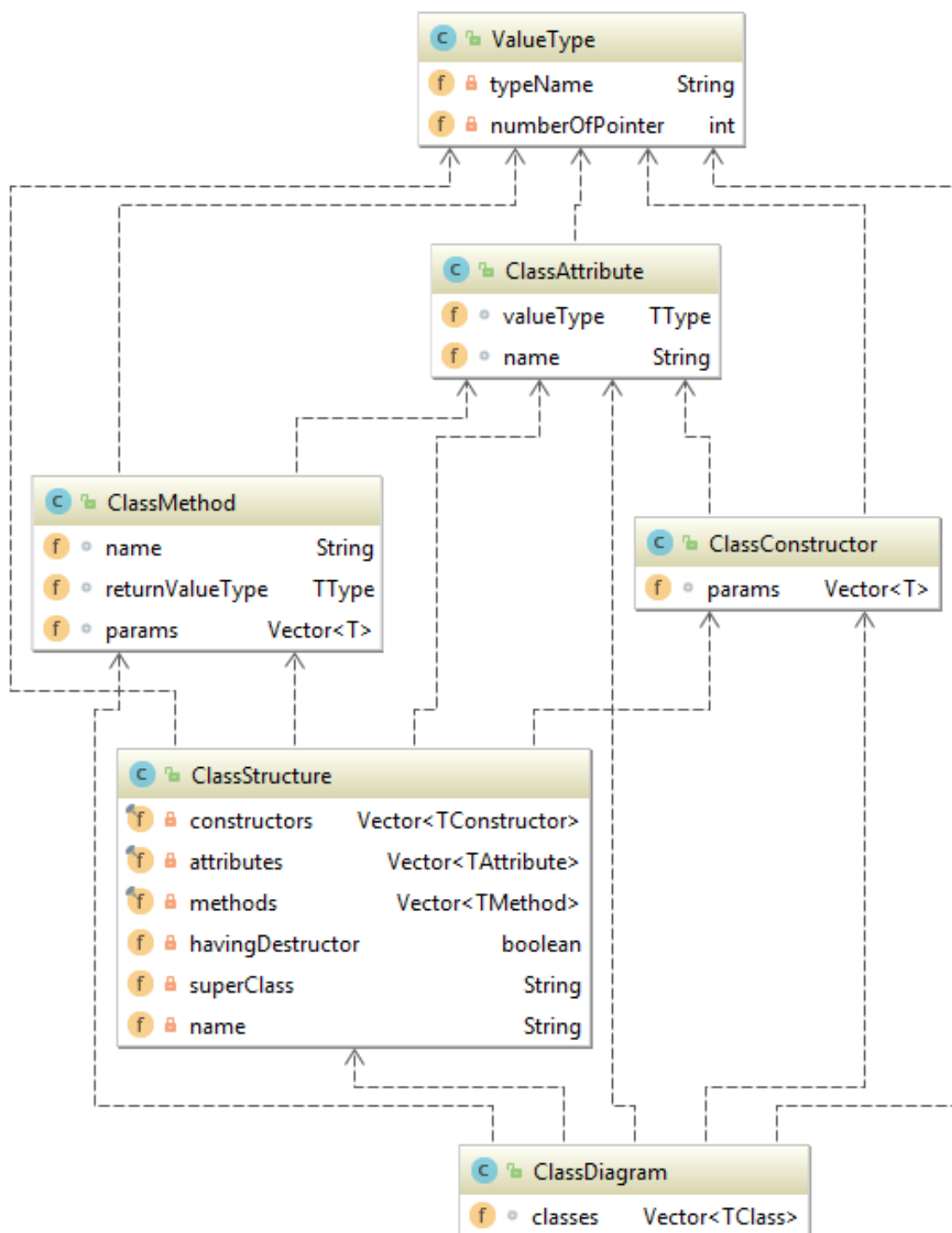
حالت دیگر گرفتن ورودی XML است.

در این حالت رشته ورودی XML به یک ساختار پرونده^{۶۸} که قابلیت تبدیل به یک نمودار کلاس را دارد تبدیل می شود. سپس از روی پرونده یک نمودار ساخته می شود. در صورتی که خطای پایه وجود نداشته باشد و وضعیت وابستگی^{۶۹} نباشد آنگاه جدول بارگذاری متدها را پاک می کند. سپس نمودار کامل برای اعمال اثربری و اضافه کردن کد برای متمایز شدن توابع ساخته می شود. بعد از بررسی دوباره خطاهای وابستگی و پایه کد فاز یک تولید می شود.

۳-۵ ساختمان اطلاعات کلاس ها

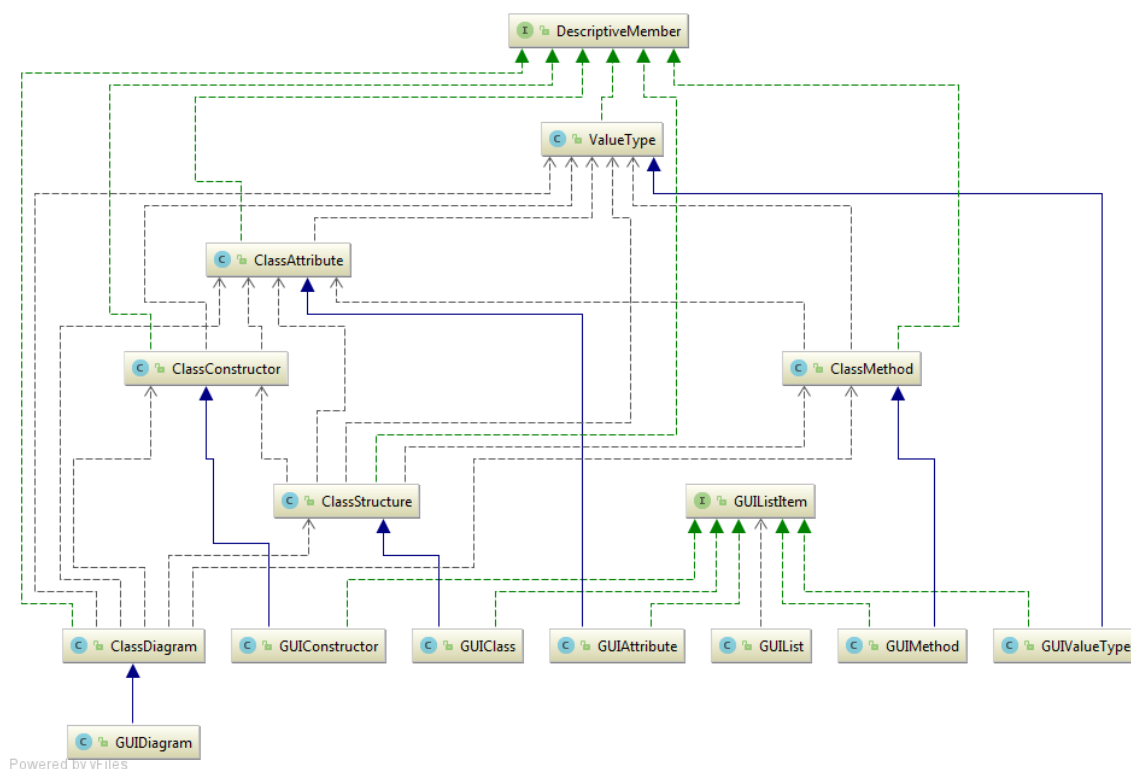
در اینجا شش ساختار پیاده سازی شده است که اطلاعات کلاس ها را ذخیره می کنند. این ساختار ها به طور جداگانه برای تولید کد و رابط کاربری گرافیکی ارث بری شده اند همچنین هر ساختار متدهایی برای خروجی دادن و گرفتن XML و بررسی خطاهای پایه دارند.

^{۶۸} Document



شکل ۴-۵: نمودار کلاس ساختمان اطلاعات کلاس ها

بخش پنجم: طراحی و پیاده سازی



شکل ۵-۵: نمودار وابستگی کلاس ها با حضور کلاس های محیط گرافیکی

DescriptiveMember ۱-۳-۵

تمام شش ساختار از این واسط^{۶۹} ارث بری کرده اند.

| <<Interface>> DescriptiveMember (com::project::classBaseUML) |
|---|
| <pre> +addDescriptionStatus(Pair<BasicDiagramStatus, LinkedList<String>>, String) : Pair<BasicDiagramStatus, LinkedList<String>> +addDescriptionStatus(Vector<Pair<BasicDiagramStatus, LinkedList<String>>>, String) : Vector<Pair<BasicDiagramStatus, LinkedList<String>>> +okStatus() : Pair<BasicDiagramStatus, LinkedList<String>> +getXml(Document) : String +getDocumentFromXML(String) : Document +getElementDocument() : Element +generateParamsTogether(Vector<?>, String...) : String +setDataByNode(Node) : void +getStatusType() : BasicDiagramStatus +statusOfMember() : Pair<BasicDiagramStatus, LinkedList<String>> +getAllProblems() : Vector<Pair<BasicDiagramStatus, LinkedList<String>>> +getShowName(String...) : String +statusOfVector(Vector<T>, boolean, String, BasicDiagramStatus) : Pair<BasicDiagramStatus, LinkedList<String>> +getAllProblemsOfVector(Vector<T>, boolean, String, BasicDiagramStatus) : Vector<Pair<BasicDiagramStatus, LinkedList<String>>> +newStatus(BasicDiagramStatus, String) : Pair<BasicDiagramStatus, LinkedList<String>> </pre> |

شکل ۵-۶: توابع واسط DescriptiveMember

^{۶۹} interface

فرزندان این واسط شش تابع را باید پیاده سازی کنند.

یک تابع جهت خروجی پرونده (تابع getElementDocumnet) است که با استفاده از آن خروجی XML تولید می شود. تابع دیگر جهت تعیین مقادیر با استفاده از پرونده (setDataByNode) است که با استفاده از آن نمودار با استفاده از رشته از XML قابل ساخت می شود. ۳ تابع دیگر برای بررسی وجود هرگونه خطای پایه ای می باشد. تابع دیگر تابع نمایش رشته ای کلاس می باشد که در تولید کد و نمایش در حالت گرافیکی کاربرد دارد.

۲-۳-۵ ValueType

این ساختار وظیفه نگهداری نوع متغیر ها را دارد.

یک Value Type دو متغیر دارد که اولی اسم نوع و دومی سطح اشاره گر است.

این ساختار وظیفه نگهداری نوع داده ها و یا نوع خروجی توابع را دارد.

```
- <Type>  
    <typename>type234</typename>  
    <number>1</number>  
</Type>
```

شکل ۷-۵ ورودی xml ساختار ValueType

۳-۳-۵ ClassAttribute

این ساختار وظیفه نگهداری متغیر ها را دارد.

یک ClassAttribute دو متغیر دارد که اسم متغیر و نوع متغیر با ساختار Value Type است.

این متغیر ها می توانند پارامتر یا مشخصه یک کلاس باشند.

| | | |
|-------------------|--------|--------|
| name: | second | |
| level of pointer: | 1 | ▲ ▼ |
| value type: | int | |

شکل ۵-۸: ورودی GUI ساختار ClassAttribute

```
- <Attribute>
    <name>new</name>
    + <Type>
</Attribute>
```

شکل ۵-۹: ورودی XML ساختار ClassAttribute

۵-۳-۴ ClassConstructor

این ساختار نماینده سازنده های کلاس ها است.

یک ClassConstructor شامل یک لیست از پارامترها با ساختار ClassAttribute می باشد.

بخش پنجم: طراحی و پیاده سازی

```
- <Constructor>
  - <params>
    + <Attribute>
    + <Attribute>
  </params>
</Constructor>
```

شکل ۵-۱۰: ورودی XML ساختار ClassConstructor

۵-۳-۵ ClassMethod

این ساختار نماینده متد های کلاس ها است.

یک ClassMethod شامل یک لیست از پارامتر ها با ساختار ClassAttribute ، اسم متغیر و نوع

خروجی تابع با ساختار Value Type می باشد.

The image shows a graphical user interface for configuring ClassMethods. It consists of two main panels, one on the left and one on the right, separated by a central vertical area.

- Left Panel:**
 - method name:** get_string
 - level of pointer:** 1 (with up/down arrows)
 - value type:** String
- Right Panel:**
 - name:** index
 - level of pointer:** 0 (with up/down arrows)
 - value type:** int
- Central Area:**
 - A vertical list titled "method int index" is positioned between the two panels.
 - At the bottom of this central area are two buttons: "add" and "delete".

شکل ۵-۱۱: ورودی GUI ساختار ClassMethod

```
- <Method>
  <name>x1</name>
  + <Type>
  - <params>
    + <Attribute>
  </params>
</Method>
```

شکل ۵-۱۲: ورودی XML ساختار ClassMethod

۵-۳-۶ ClassStructure

این ساختار نماینده یک کلاس در چیدمان ما می باشد.

یک ClassStructure شامل یک لیست از مشخصه ها با ساختار ClassAttribute، یک لیست از سازنده ها با ساختار ClassConstructor، یک لیست از متد ها با ساختار ClassMethod، یک متغیر دودویی برای تعیین داشتن تخریب گر، اسم متغیر و اسم کلاس پدر (که در صورت نبودن پدر مقدار متغیر رشته null است) می باشد.

```
- <Class>
  <name>Student</name>
  <super>Object</super>
  <destructor>true</destructor>
  - <attributes>
    + <Attribute>
    + <Attribute>
  </attributes>
  - <constructors>
    + <Constructor>
    + <Constructor>
  </constructors>
  - <methods>
    + <Method>
    + <Method>
  </methods>
</Class>
```

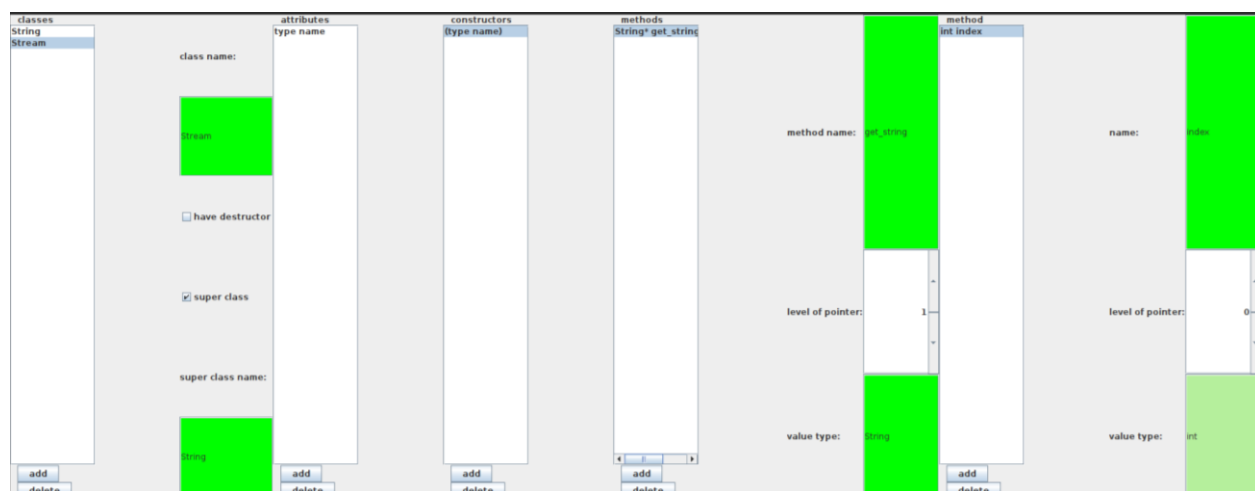
شکل ۵-۱۳: ورودی XML ساختار ClassStructure

۵-۳-۷ ClassDiagram

این ساختار نمودار کلاس را در خود ذخیره می کند.

ClassDiagram شامل یک لیست از کلاس ها با ساختار ClassStructure می باشد.

بخش پنجم: طراحی و پیاده سازی



شکل ۵-۱۴: ورودی GUI ساختار ClassDiagram

```
- <ClassDiagram>
+ <Class>
+ <Class>
+ <Class>
+ <Class>
</ClassDiagram>
```

شکل ۵-۱۵: ورودی GUI ساختار ClassDiagram

۵-۴ خطاهای پایه

اطلاعات داده شده توسط کاربر به از نظر صحت و درستی به دو روش چک می شوند. در روش اول که خطاهای پایه ای هستند هدف ما پیدا کردن خطاهایی هستند که کامپایلر خواهد گرفت و برای همین برنامه در صورت وقوع این نوع خطاها اجازه ادامه کار را نمی دهد. انواع خطاها بر اساس اسم در ادامه شرح داده خواهد شد.

همچنین تمامی اسامی باید از الگوی زیر پیروی کنند:

$$*[_a - zA - Z][_a - zA - Z0 - 9]$$

SameConstructor -۱

در این خطا یک کلاس دارای دو سازنده یکسان هست. دو سازنده در صورتی برابرند که ValueType های هر دو لیست پارامترهای سازنده ها با هم برابر باشند.

SameMethodSignature -۲

در این خطا یک کلاس دارای دو متد هم امضا هست. دو متد در صورتی هم امضاء هستند که ValueType های هر دو لیست پارامترها و اسم متد ها با هم برابر باشند.

SameAttributeName -۳

در این خطا یک کلاس دارای دو مشخصه با اسم یکسان است.

SameAttributeAndMethodName -۴

در این خطا یک کلاس دارای یک Attribute و یک متد با اسم یکسان است.

SameAttributeAndClassName -۵

در این خطا یک کلاس دارای یک Attribute هم اسم با یکی از کلاس های نمودار است.

SameMethodAndClassName -۶

در این خطا یک کلاس دارای یک متد هم اسم با یکی از کلاس های نمودار است.

SameClassName -۷

در این خطا دو کلاس هم اسم هستند.

SameParameterName -۸

در این خطا دو پارامتر در متد و یا سازنده با هم برابرند.

۹- NegativeValueTypePointer

تعداد پوینتر های یک نوع نمی تواند منفی باشد و این خطا برای این حالت می باشد. البته این خطا فقط در ورودی XML ممکن است به وجود بیاید.

۱۰- TypeNameError

در صورتی که اسم یک نوع اشکال داشته باشد این خطا را شاهدیم. اسم نوع از کلمات ذخیره شده نباید باشد و باید از الگوی مربوطه تبعیت کند.

۱۱- AttributeNameError

در صورتی که اسم یک مشخصه اشکال داشته باشد این خطا را شاهدیم. اسم مشخصه از کلمات ذخیره شده و نوع ها و توصیف کننده انواع نباید باشد و باید از الگوی اسم تبعیت کند. اسم مشخصه می تواند شامل درایه ها باشد.

۱۲- MethodNameError

در صورتی که اسم یک متد اشکال داشته باشد این خطا را شاهدیم. اسم متد از کلمات ذخیره شده و نوع ها و توصیف کننده انواع نباید باشد و باید از الگوی اسم تبعیت کند.

۱۳- ClassNameError

در صورتی که اسم یک کلاس اشکال داشته باشد این خطا را شاهدیم. اسم کلاس از کلمات ذخیره شده و نوع ها و توصیف کننده انواع نباید باشد و باید از الگوی اسم تبعیت کند.

۱۴- SuperClassNameError

در صورتی که اسم پدر یک کلاس اشکال داشته باشد این خطا را شاهدیم. اسم پدر کلاس از کلمات ذخیره شده و نوع ها و توصیف کننده انواع نباید باشد و باید از الگوی اسم تبعیت کند.

۱۵- NonExistSuperClass

در صورتی که اسم پدر در لیست کلاس ها نباشد این خطا را شاهدیم.

۵-۵ وابستگی کلاس ها

۳ وابستگی تعریف شده کلاس ها و حلقه های وابستگی و عملکرد نرم افزار را بعد از بدست آوردن حلقه های تعریف شده در نمودار در این قسمت توضیح داده می شوند.

۵-۵-۱ وابستگی نوع *

کلاس a به کلاس b وابستگی به نوع * دارد اگر بدون ساخته شدن کلاس b ساخته شدن کلاس a غیر ممکن باشد. دو نوع وابستگی نوع * را با اسم شرح می دهیم.

۱- SuperClass

این وابستگی در صورتی به وجود می آید که یک کلاس تعمیم کلاس دیگر باشد. کلاس فرزند تمام ویژگی های پدر به جز سازنده را دارد و عملاً وابسته به ساخته شدن آن است. به طور مثال کلاس a و b نمی توانند پدر همدیگر باشند.

۲- ObjectAttribute

این وابستگی زمانی به وجود می آید که یک کلاس یک شی از کلاس دیگر را داشته باشد. مشخصه های یک کلاس باید قبل از اتمام کار سازنده و یا اختصاص حافظه به آن ساخته شوند و اگر مشخصه اشاره گر به حافظه نباشد کلاس به نوع مشخصه وابسته است. به طور مثال کلاس a و b نمی توانند عضوی از همدیگر داشته باشند.

۳- ConstructorAllObjectParameter

این وابستگی زمانی به وجود می آید که در تمام سازنده های یک کلاس شی کلاس خاصی به عنوان پارامتر ظاهر بشود.

اگر تمام سازنده های کلاس a دارای پارامتری از نوع کلاس b باشد، می توانیم نتیجه بگیریم ساخته شدن کلاس a وابسته به ساخته شدن کلاس b می باشد.

۵-۵-۲ وابستگی نوع ۱

کلاس a به کلاس b وابستگی به نوع ۱ دارد اگر بدون ساخته شدن کلاس b، ساخته شدن کلاس a در شرایط خاصی ممکن باشد. دو نوع وابستگی نوع ۱ را با اسم شرح می دهیم.

۱- ConstructorAllPointerParameter

این وابستگی زمانی به وجود می آید که در تمام سازنده های یک کلاس اشاره گر کلاس خاصی به عنوان پارامتر ظاهر بشود.

۲- ConstructorAllHybridParameter

این وابستگی زمانی به وجود می آید که در تمام سازنده های یک کلاس اشاره گر و شی کلاس خاصی به عنوان پارامتر ظاهر بشود.

این گروه وابستگی شرایط خاصی را بین دو کلاس ایجاد می کند. اگر به طور مثال کلاس a یک مشخصه در کلاس b داشته باشد و کلاس a وابستگی نوع ۱ به کلاس b داشته باشد، پارامتر اشاره گر به کلاس b اولین سازنده ای که از a برای ساخته شدن صدا زده می شود در عمل باید به null اشاره کند.

همانطور که می بینید وابستگی نوع ۱ یک کلاس به کلاس دیگر از ساخته شدن قبل از آن جلوگیری نمی کند ولی نحوه ساخت آن مورد سوال خواهد بود.

۵-۵-۳ وابستگی نوع ۲

کلاس a به کلاس b وابستگی به نوع ۲ دارد اگر بدون ساخته شدن کلاس b، صدا زدن یکی از متدها و یا سازنده ها در کل مجموعه غیر ممکن باشد و یا باید از اشاره گر هایی استفاده کند که به null اشاره می کنند.

۱- PointerAttribute

این وابستگی زمانی به وجود می آید که یک کلاس یک اشاره گر از کلاس دیگر را به عنوان مشخصه داشته باشد.

۲- MethodObjectType

این وابستگی زمانی به وجود می آید که نوع خروجی یک متد، شی کلاس دیگری باشد.

۳- MethodPointerType

این وابستگی زمانی به وجود می آید که نوع خروجی یک متد، اشاره گر به شی کلاس دیگری باشد.

۴- MethodObjectParameter

این وابستگی زمانی به وجود می آید که پارامتر یک متد، شی کلاس دیگری باشد.

۵- MethodPointerParameter

این وابستگی زمانی به وجود می آید که پارامتر یک متد، اشاره گر شی کلاس دیگری باشد.

۶- ConstructorObjectParameter

این وابستگی زمانی به وجود می آید که پارامتر یک سازنده، شی کلاس دیگری باشد.

۷- ConstructorPointerParameter

این وابستگی زمانی به وجود می آید که پارامتر یک سازنده، اشاره گر شی کلاس دیگری باشد.

۵-۵-۴ حلقه وابستگی نوع ۰

حلقه وابستگی نوع ۰ زمانی اتفاق می افتد که چند کلاس که به هم وابستگی نوع ۰ دارند یک دور را تشکیل بدهند. در این صورت هیچ کدام از این کلاس ها نمی توانند ساخته شوند و نرم افزار اجازه ادامه کار را نمی دهد.

۵-۵-۵ حلقه وابستگی نوع ۱

حلقه وابستگی نوع ۱ زمانی اتفاق می افتد که چند کلاس که به هم وابستگی نوع ۰ و یا نوع ۱ دارند یک دور را تشکیل بدهند. کلاس ها در این شرایط می توانند ساخته شوند ولی حداقل یک سازنده دارای پارامتری می باشد که اشاره گر به null است و بعد تا قبل از اتمام سازنده این اشاره گر null می ماند. که غیر طبیعی است. در صورت پیدا شدن حلقه وابستگی نوع ۱ برنامه به کاربر اخطار می دهد و سپس اجازه ادامه کار را می دهد.

۵-۵-۶ حلقه وابستگی نوع ۲

حلقه وابستگی نوع ۲ زمانی اتفاق می افتد که چند کلاس که به هم وابستگی نوع ۰ و یا نوع ۱ و یا نوع ۲ دارند یک دور را تشکیل بدهند. وجود این حلقه وابستگی طبیعی است. به طور مثال در پیاده سازی لیست پیوندی^{۷۰} هر کلاس در خود دو اشاره گر از جنس خود کلاس را دارد که یکی از آن ها به عنصر بعدی کلاس و دیگری به عنصر قبلی کلاس اشاره می کند. با وجود منطقی بودن وجود حلقه نوع ۲، در صورت پیاده سازی اشتباه امکان به وجود آمدن یک حلقه بی نهایت و در نتیجه اتمام برنامه به خاطر خطای حافظه ای وجود دارد. در صورت پیدا شدن حلقه وابستگی نوع ۲، وجود این حلقه به کاربر اطلاع داده می شود و سپس برنامه به کار خود ادامه می دهد.

^{۷۰} Linked List

۷-۵-۵ پیدا کردن حلقه وابستگی

ما از الگوریتم زیر برای پیدا کردن وابستگی استفاده می کنیم:

```
findMostCycles()
{
    dependencyNumber = 3;
    for (String name:classNames)
        for(int Type = 0; Type < name.DependencyCycleType; Type++)
            for(String node:classNames)
                node.reset();
    dfsResult = dfs(name,name,Type);
    if (dfsResult != null)
        allCycles[Type].add(dfsResult);
    dependencyNumber = min(dependencyNumber, Type);
}

dfs(v, start, Type)
{
    if (v == start && v.Visit())
        return [];
    if (v.Visit())
        return null;
    v.setVisit(true);
    for(edge:v.getEdges(=<Type))
        dependencyEdges = dfs(edge.getFatherNode(), start, Type);
    if(dependencyEdges != null)
        v.setDependencyCycleType(Type);
        dependencyEdges.add(edge);
        return dependencyEdges;
    return null;
}
```

شکل ۵-۱۶: الگوریتم پیدا کردن دور های وابستگی

در الگوریتم بالا ما تلاش داریم که اگر کلاسی در دوری وجود دارد، آن دور اضافه شود.

همچنین به دنبال دور های با وابستگی نوع های پایین تر هستیم زیرا نوع های وابستگی های پایین تر خطرناک تر هستند و در صورت وجود دور ۰ نمودار ایراد دارد.

۵-۶ تولید کد ابتدایی در مرحله اول

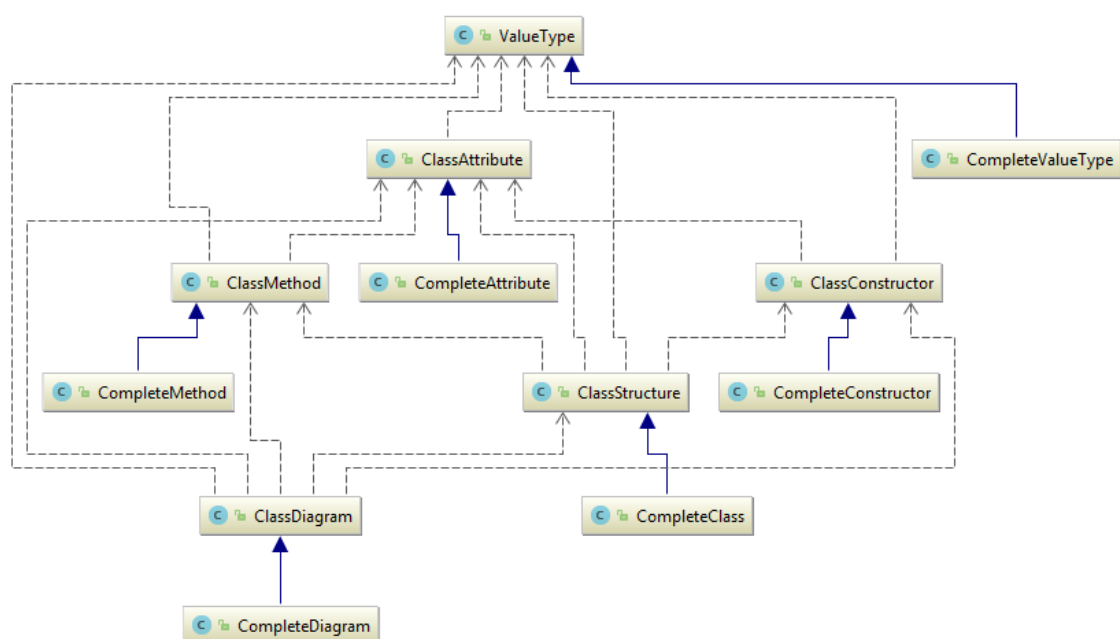
مراحل تولید کد به ترتیب ارائه شده می باشد.

۵-۶-۱ اعمال ارث بری

بعد از اینکه اطلاعات گرفته شده از کاربر بررسی شد، ارث بری ها اعمال می شوند به نحوی که هر کلاس مشخصه ها و متد های غالب نشده کلاس پدر خود را می گیرد.

بعد از اعمال ارث بری دوباره خطاهای پایه و حلقه های وابستگی بررسی می شوند و در صورتی که پیدا شد برنامه ادامه کار را قطع می کند.

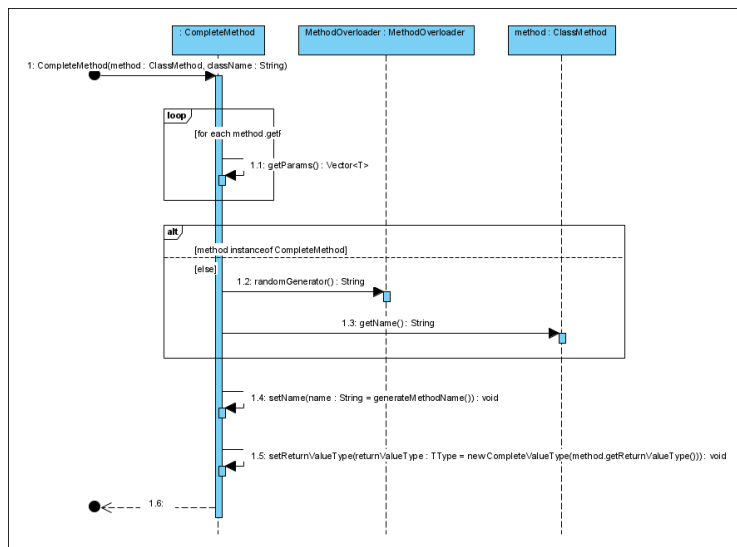
اعمال این ارث بری با استفاده از کلاس ساختمان های جدیدی صورت می گیرد که ساختمان جدید به شکل زیر است.



شکل ۵-۱۷: ساختمان برای تولید کد

در این مرحله هر ۶ ساختار جدید در موقع ساخته شدن، یک ساختار قدیم را ورودی می گیرند.

بخش پنجم: طراحی و پیاده سازی



شکل ۵-۱۸: ساختمان متد برای تولید کد

۵-۶-۲ تعریف ساختار

ابتدا به هر کلاس یک مشخصه به نام `this` اختصاص داده می شود که در آن آدرس خود شی ذخیره می شود. سپس هر کلاس را ما در قالب `union` به شکل زیر ذخیره می کنیم:

```

union DefinedClass
{
    union Parent1 unionParent1;
    union Parent2 unionParent2;
    union Parent3 unionParent3;
    .
    .
    .

    struct
    {
        union DefinedClass* this;
        Attribute1Type attribute1Name;
        Attribute2Type attribute2Name;
        Attribute3Type attribute3Name;
        .
        .
        .
    };
};
    
```

شکل ۵-۱۹: قالب ذخیره سازی ساختار یک کلاس

در شکل بالا تمام مشخصه ها در داخل struct قرار می گیرند و تمام ساختمان های پدر کلاس حافظه را به صورت اشتراکی دارند.

در عمل با فراخوانی `a.unionParentx` ما `a` را به `Parentx`، تبدیل به بالا کرده ایم.

۵-۶-۳ تعریف متدها

در تعریف هر متد (چه عادی، چه سازنده و چه تخریب گر) پارامتر اشاره گر خود کلاس اضافه می شود و به مانند تابع با آن عمل می گردد.

همچنین در فایلی دیگر از کاربر می خواهیم پیاده سازی متدها را انجام بدهد.

همچنین برای هر سازنده یک تابع جهت `new` کردن وجود دارد که اشاره تابع ساخته شده را بر می گرداند. تابع `delete` برای تمام کلاس ها ساخته شده تا حافظه مصرف شده را آزاد کند. در صورت داشتن تخریب اگر قبل از آزاد سازی حافظه تابع تخریب گر صدا زده می شود.

```
Student::Student(int number_of_professors)
{
    // TODO:code here
}

Student::Student(class FixedSizeString the_name, int number_of_professors)
{
    // TODO:code here
}

class FixedSizeString Student::get_professor(int number)
{
    // TODO:code here
}

void Student::set_professor(int number, class FixedSizeString professor_name)
{
    // TODO:code here
}

Student::~~Student()
{
    // TODO:code here
}
```

شکل ۵-۲۰: فایل تعریف برای پیاده سازی متدها

بخش پنجم: طراحی و پیاده سازی

```
class FixedSizeString get_professor(union Student* this, int number);
void set_professor(union Student* this, int number, class FixedSizeString professor_name);

void constructorStudent(union Student* this, int number_of_professors);
union Student* newStudent(int number_of_professors);
void constructorStudent(union Student* this, class FixedSizeString the_name, int number_of_professors);
union Student* newStudent(class FixedSizeString the_name, int number_of_professors);

void destructorStudent(union Student* this);
void delete(union Student* this);
```

شکل ۵-۲۱: تعریف متد های برای یک کلاس

همچنین در صورت صدا زده شدن متد های کلاس پدر، با استفاده از قابلیت تبدیل به بالای ارائه شده تابع کلاس پدر صدا زده می شود.

```
void set_chars(union FixedSizeString* this, char* characters, int size_of_string)
{
    set_chars(&(this->unionString), characters, size_of_string);
}
char* get_chars(union FixedSizeString* this)
{
    return get_chars(&(this->unionString));
}

union FixedSizeString* newFixedSizeString()
{
    union FixedSizeString* this = (union FixedSizeString*) malloc(sizeof(union FixedSizeString));
    constructorFixedSizeString(this);
    return this;
}

union FixedSizeString* newFixedSizeString(int size_of_string)
{
    union FixedSizeString* this = (union FixedSizeString*) malloc(sizeof(union FixedSizeString));
    constructorFixedSizeString(this, size_of_string);
    return this;
}

void delete(union FixedSizeString* this)
{
    destructorFixedSizeString(this);
    free(this);
}
```

شکل ۵-۲۲: پیاده سازی متد های پدر و توابع تخصیص حافظه برای یک کلاس

البته در سه شکل بالا دستکاری نام را نشان نداده ایم. با استفاده از روش های توضیح داده شده در بخش ۲-۴-۱ اسامی توابع دستکاری می شوند و سپس برای استفاده یک سربار گذاری مصنوعی را در برنامه به وجود می آوریم.

۵-۶-۴ اطلاعات پایه ای کلاس ها

اطلاعات پایه ای کلاس ها که شامل اسم آن ها ، مشخصه ها، اسم متد ها، اسم کلاس های پدر و دارا بودن سازنده و تخریب گر برای استفاده در مرحله دوم تولید کد خروجی داده می شوند.

۵-۷-۷ تولید کد نهایی در مرحله دوم

در اینجا تمام فایل های مرحله اول که توسط کاربر تغییر داده شده و اطلاعات پایه ای کلاس ها به عنوان ورودی گرفته شده اند.

۵-۷-۱ تحلیل لغوی

در این مرحله تمام نشانه ها از فایل های سی خوانده توسط کد نوشته در زبان پایتون خوانده می شوند. این کد از بخش LEX کتابخانه PLY استفاده کرده و نشانه های مورد نیاز و نشانه های خود زبان سی را استخراج می کند.

۵-۷-۲ ترنسکامپایل کد به سی

در این مرحله برای هر فایل لیستی از نشانه ها داریم.

در این مرحله برنامه کار های زیر را انجام می دهد:

۱- تعریف متد های تحویل شده به کاربر را تبدیل به تابع می کند.

۲- متد های صدا زده شده را به توابع مورد نظرشان ارجاع می دهد.

۳- در زمان صدا زدن سازنده، مقدار کلاس تعیین می شود.

۴- در زمان خروج از محدوده کد تخریب گر را صدا می زند.

در ارزیابی می خواهیم بررسی کنیم آیا برنامه موفق بوده است یا خیر.
به طور کلی می توان توانایی های برنامه را به ۵ بخش مهم تقسیم کرد:

- ۱- پیدا کردن خطاهای پایه
- ۲- پیدا کردن خطاهای وابستگی
- ۳- رابط گرافیکی کاربری
- ۴- تولید کد در بخش اول
- ۵- ترنسکامپایل کد در بخش دوم

سه قابلیت اول بسیار کاربردی هستند و نوشتن آزمون برای آن ها قطعاً کار مفیدی است ولی برنامه بدون وجود این سه بخش با فرض عملکرد درست کاربر ما یک برنامه کامل را خواهیم داشت.
پس برای ارزیابی آزمون هایی را برای دو بخش آخر ترتیب خواهیم داد.

یک آزمون به این شکل است که یک نمودار در مرحله اول تولید شده است و کد خروجی آن تغییر داده شده است را به مرحله دوم داده می شود و خروجی استاندارد آن ^{۷۱} با برنامه معادل آن در زبان سی پلاس پلاس مقایسه می شود. در صورت برابر بودن خروجی نتیجه تست موفق ثبت می شود.

نیازمندی های اصلی که در شرح پروژه آمده اند به شرح زیر هستند:

- ۱- ارثیری
- ۲- متد
- ۳- چندریختی
- ۴- سازنده
- ۵- تخریب گر

^{۷۱} stdout

نیازمندی های دیگری هم بررسی خواهند شد که وابسته به نیازمندی های بالا هستند و یا در صورت برآورده نشدن آن ها پروژه به هدف خود نرسیده است.

۶- سربارگذاری تابع delete: این نیازمندی عملاً ترکیبی از نیاز به تخریب گر و چند ریختی است.

این نیازمندی برای شرایطی است که بیشتر از یک کلاس با تخریب گر داشته باشیم.

۷- سربارگذاری تابع سازنده: این نیازمندی عملاً ترکیبی از نیاز به سازنده و چند ریختی است. این

نیازمندی برای شرایطی است که در یک کلاس بیشتر از یک سازنده داشته باشیم.

۸- سربارگذاری متد: این نیازمندی عملاً ترکیبی از نیاز به متد و چند ریختی است. این نیازمندی برای

شرایطی است که چند متد با اسم یکسان داشته باشیم.

۹- غالب کردن متد: این نیازمندی عملاً ترکیبی از نیاز به متد و ارثیری است. این نیازمندی برای

شرایطی است که کلاس فرزند دارای متدی هم اسم و هم امضا با پدرش داشته باشیم.

۱۰- تبدیل به نوع بالا: این نیازمندی مربوط به ارثیری است و در ارثیری متد و استفاده از توابع به ما کند

می کند.

۱۱- استفاده یک کلاس در کلاس دیگر: کلاس ها باید بتوانند اعضای یکدیگر را داشته باشند و در

توابع به عنوان ورودی بگیرند و آن ها را خروجی بدهند.

۱۲- استفاده از آرایه: باید بتوان یک کلاس با یک آرایه ساخت.

۱۳- عدم تاثیر در بخش های نامربوط کد: برنامه نباید کد های عادی سی مانند پیش پردازنده ها و

نامربوط به کلاس ها را دستکاری بکند.

حال در مجموع ۱۵ آزمون برای برنامه آماده شده است که حالت های متفاوتی از نیازمندی ها را پوشش می

دهد.

در اینجا پنج آزمون که پوشش دهنده قابلیت های بالاست توضیح داده خواهد شد.

آزمون صفر: این آزمون صرفاً یک کد به زبان سی می باشد. هدف این آزمون فراهم کردن نیازمندی

سیزدهم یعنی عدم تاثیر برنامه در بخش های نامربوط به قابلیت هایی است که پروژه فراهم کرده است.

آزمون اول:

| Empty |
|---|
| +Empty() +Empty(i : int) +Empty() |

شکل ۶-۱: نمودار آزمون اول

در این آزمون ما می خواهیم که چهار قابلیت را بررسی کنیم:

۱- چندریختی (۳)

۲- قابلیت داشتن سازنده (۴)

۳- داشتن تخریبگر (۵)

۴- سربارگذاری تابع سازنده (۷)

در داخل کد تولیدی برای سربارگذاری سازنده از تعداد ورودی آن استفاده شده است.

```
#define newEmpty0(...) \
newEmpty_KVNAQsecw \
(__VA_ARGS__)
#define newEmpty1(...) \
newEmpty_k8biul8f4 \
(__VA_ARGS__)

#define SELECT_N(X, _1, _2, _3, _4, N, ...) N

#define constructorEmpty(...) SELECT_N(X, ##__VA_ARGS__, constructorEmpty4,
constructorEmpty3, constructorEmpty2, constructorEmpty1,
constructorEmpty0)(__VA_ARGS__)
#define delete_keyword(...) SELECT_N(X, ##__VA_ARGS__, delete_keyword4,
delete_keyword3, delete_keyword2, delete_keyword1, delete_keyword0)(__VA_ARGS__)
#define newEmpty(...) SELECT_N(X, ##__VA_ARGS__, newEmpty4, newEmpty3, newEmpty2,
newEmpty1, newEmpty0)(__VA_ARGS__)
```

شکل ۶-۲: بخشی از کد خروجی مرحله اول آزمون اول

در بالا برای هر تابع بر اساس تعداد پارامترها جدا می شوند و سپس توابع جدا شده تعریف می شوند.

```
#include <stdio.h>
#include "overload.h"

int testNonPointer()
{
    class Empty e1(12), e2();
    return 7;
}

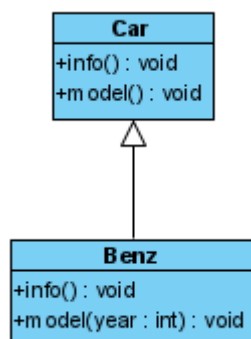
void testWithPointer()
{
    class Empty* e3;
    e3 = new Empty(4);
    delete(e3);
}

int main()
{
    testNonPointer();
    testWithPointer();
}
```

شکل ۶-۳: کد استفاده شده در آزمون اول

در بالا حالات متفاوت سازنده و تخریبگر تست شده اند.

آزمون دوم:



شکل ۶-۴: نمودار آزمون دوم

بخش ششم: ارزیابی

در این آزمون ما می خواهیم که چهار قابلیت را بررسی کنیم:

- ۱- ارثیری (۱)
- ۲- متد (۲)
- ۳- سربار گذاری متد (۸)
- ۴- غالب کردن متد (۹)

برنامه برای ارث بردن متد و همچنین سربار گذاری مانند آزمون اول کدی با استفاده پیش پردازنده ها تولید کرده است ولی در اینجا توابعی با یک اسم و یک تعداد پارامتر داریم.

```
#define CHOOSE __builtin_choose_expr
#define IFTYPE(X, T) __builtin_types_compatible_p(typeof(X), T)

#define SELECT_1(X1, ...) X1
#define SELECT_2(X1, X2, ...) X2
#define SELECT_3(X1, X2, X3, ...) X3
#define SELECT_4(X1, X2, X3, X4, ...) X4

#define model1(...) \
    CHOOSE(IFTYPE(SELECT_1(__VA_ARGS__), union Benz*), model_BUb2bJfj3, \
    model_0ZgXXBEWz) \
    (__VA_ARGS__)
#define model2(...) \
    model_C9PoYoVFF \
    (__VA_ARGS__)

#define info1(...) \
    CHOOSE(IFTYPE(SELECT_1(__VA_ARGS__), union Benz*), info_0klPTXPmp, \
    info_lH0778d69) \
    (__VA_ARGS__)

#define SELECT_N(X, _1, _2, _3, _4, N, ...) N

#define model(...) SELECT_N(X, ##__VA_ARGS__, model4, model3, model2, model1, model0)(__VA_ARGS__)
#define info(...) SELECT_N(X, ##__VA_ARGS__, info4, info3, info2, info1, info0)(__VA_ARGS__)
```

شکل ۵-۶: بخشی از کد خروجی سرآیند مرحله اول آزمون دوم

در بالا با استفاده از نوع پارامتر تابع تعیین می شود.

```
union Benz
{
    union Car unionCar;
    struct
    {
        union Benz* this;
    };
};

void model_BUb2bJfj3(union Benz* this);
void info_0klPTXPmp(union Benz* this);
void model_C9PoYoVFF(union Benz* this, int year);

void model_BUb2bJfj3(union Benz* this)
{
    model(&(this->unionCar));
}
```

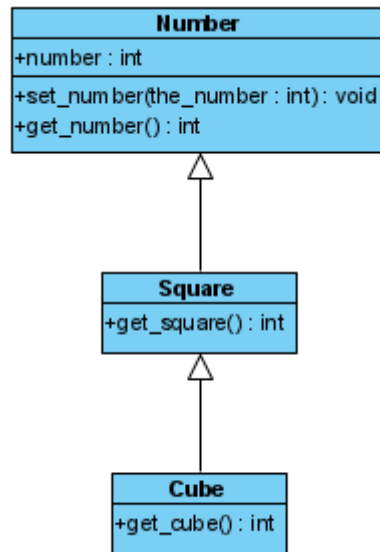
شکل ۶-۶: بخشی از کد خروجی پیاده سازی شده مرحله اول آزمون دوم

در شکل بالا تابع خروجی داده شده برای به ارث بردن تابع `model` بدون پارامتر نشان داده شده است که از تبدیل به بالا استفاده می کند.

```
int main()
{
    class Car car;
    class Benz benz;
    car.info();
    car.model();
    benz.model(2007);
    benz.info();
    benz.model();
    benz.model(2019);
}
```

شکل ۶-۷: کد استفاده شده در آزمون دوم

آزمون سوم:



شکل ۶-۸: نمودار آزمون سوم

در این آزمون ما می خواهیم که سه قابلیت را بررسی کنیم:

- ۱- ارثیری (۱): با این تفاوت که بیشتر از یک لایه درگیر است.
- ۲- سربارگذاری متد (۸): با این تفاوت که در اینجا پارامتر هم داریم.
- ۳- تبدیل به نوع بالا (۱۰)

```

#define set_number2(...) \
    CHOOSE(IFTYPE(SELECT_1(__VA_ARGS__), union Square*) && IFTYPE(SELECT_2(__VA_ARGS__), int), set_number_PciYH08yM, \
    CHOOSE(IFTYPE(SELECT_1(__VA_ARGS__), union Cube*) && IFTYPE(SELECT_2(__VA_ARGS__), int), set_number_2pAFupUyz, \
    set_number_fd2pJn)) \
    (__VA_ARGS__)

#define get_number1(...) \
    CHOOSE(IFTYPE(SELECT_1(__VA_ARGS__), union Square*), get_number_HhqFwykBD, \
    CHOOSE(IFTYPE(SELECT_1(__VA_ARGS__), union Cube*), get_number_Iml1UPzTF, \
    get_number_GVyIj18aj)) \
    (__VA_ARGS__)

#define SELECT_N(X, _1, _2, _3, _4, N, ...) N

#define get_cube(...) SELECT_N(X, ##__VA_ARGS__, get_cube4, get_cube3, get_cube2, get_cube1, get_cube0)(__VA_ARGS__)
#define get_square(...) SELECT_N(X, ##__VA_ARGS__, get_square4, get_square3, get_square2, get_square1, get_square0)(__VA_ARGS__)
#define set_number(...) SELECT_N(X, ##__VA_ARGS__, set_number4, set_number3, set_number2, set_number1, set_number0)(__VA_ARGS__)
#define get_number(...) SELECT_N(X, ##__VA_ARGS__, get_number4, get_number3, get_number2, get_number1, get_number0)(__VA_ARGS__)
    
```

شکل ۶-۹: بخشی از کد خروجی سرآیند مرحله اول آزمون دوم

در شکل بالا شاهد هستیم در صورتی که پارامتری داشته باشیم تمام پارامترها چک خواهند شد.

```
union Cube
{
    union Number unionNumber;
    union Square unionSquare;
    struct
    {
        union Cube* this;
        int number;
    };
};

int get_number_Im1lUPzTF(union Cube* this);
void set_number_2pAFupUyz(union Cube* this, int the_number);
int get_square_XNMuXshEz(union Cube* this);
int get_cube_cxz4WH7zQ(union Cube* this);

int get_number_Im1lUPzTF(union Cube* this)
{
    return get_number(&(this->unionNumber));
}

void set_number_2pAFupUyz(union Cube* this, int the_number)
{
    set_number(&(this->unionNumber), the_number);
}

int get_square_XNMuXshEz(union Cube* this)
{
    return get_square(&(this->unionSquare));
}
```

شکل ۶-۱۰: بخشی از کد خروجی پیاده سازی شده مرحله اول آزمون سوم

در شکل بالا می توان دید که امکان تغییر مستقیم به کلاس های بالاتر از پدر وجود دارد و در صورتی که متد ارث رسیده بالاتر از پدر باشد مستقیم آن تابع را صدا می کند و از تعداد دفعات صدا زدن تابع جلوگیری می نماید.

```
int main()
{
    class Cube cube;
    cube.set_number(7);
    printf("%d\n", cube.get_number());
    printf("%d\n", cube.get_square());
    printf("%d\n", cube.get_cube());

    class Square square = cube.unionSquare;
    printf("%d\n", square.get_number());
    printf("%d\n", square.get_square());
    square.set_number(5);
    printf("%d\n", square.get_number());
    printf("%d\n", square.get_square());

    class Number number_1 = cube.unionNumber;
    printf("%d\n", number_1.get_number());

    class Number number_2 = square.unionNumber;
    printf("%d\n", number_2.get_number());
}
```

شکل ۶-۱۱: کد استفاده شده در آزمون سوم

آزمون چهارم:



شکل ۶-۱۲: نمودار آزمون چهارم

در این آزمون ما می خواهیم که دو قابلیت را بررسی کنیم:

۱- استفاده یک کلاس در کلاس دیگر (۱۱)

۲- استفاده از آرایه (۱۲)

```
union Map
{
    struct
    {
        union Map* this;
        class Point points[8];
    };
};

void set_point_6xLE8tyZB(union Map* this, int index, int x, int y);
class Point get_point_pJ0cdCxyo(union Map* this, int index);
```

شکل ۶-۱۳: بخشی از کد خروجی پیاده سازی شده مرحله اول آزمون چهارم

در شکل بالا استفاده از کلاس Point را در کلاس Map داریم و تعریف متد را در زیر آن می بینم.

```
void Map::set_point_6xLE8tyZB(int index, int x, int y)
{
    (this->points[index]).set(x, y);
}

class Point Map::get_point_pJ0cdCxyo(int index)
{
    return this->points[index];
}
```

شکل ۶-۱۴: پیاده سازی از سمت کاربر برای متدهای تعریف شده در آزمون چهارم

در شکل بالا در اولین تابع، متد یکی از خانه ی آرایه صدا زده می شود تا ۲ مقدار آن را تعیین کند. در تابع دوم مقدار یک آرایه از جنس کلاس را خروجی می دهد.

```
int main()
{
    class Map our_map;
    our_map.set_point(3, 4, 5);
    our_map.set_point(2, 3, 8);
    our_map.set_point(1, 2, 5);
    class Point third;
    third = our_map.get_point(3);
    printf("%d\n", third.x);
    printf("%d\n", our_map.get_point(2).y);
}
```

شکل ۶-۱۵: کد استفاده شده در آزمون چهارم

بخش هفتم: نتیجه گیری

فصل هفتم

نتیجه گیری

۷-۱ جمع بندی

در این پروژه سعی شده است استفاده قابلیت های مهمی از شی گرای را در زبان سی ممکن و راحت تر بسازد. قابلیت داشتن ارث بری، سازنده، تخریب گر، سربار گذاری، قابل کردن متد و تبدیل به بالا قابلیت های اصلی یک زبان شی گرا هستند که در این پروژه ارائه شد. با اینحال چند ریختی پارامتری و محدود کردن دسترسی جزو قابلیت های زیادی بودند که در این پروژه پیاده سازی نشد. همچنین برنامه به پیش پردازنده زبان سی حساسیت ندارد که نکته منفی آن است ولی باید توجه داشت که بدون دسترسی به محیط و تغییر آن (که متضاد با ذات پروژه بود) دانستن خروجی برنامه بعد از اعمال پیش پردازنده ها غیر ممکن است.

این پروژه به صورت متن باز در آدرس زیر در دسترس می باشد.

https://github.com/bigsheykh/Convert_UML_to_ANSI_C

۷-۲ چالش ها و کارهای آینده

در این پروژه چالش های متفاوتی پیش رو بود. تعدادی از آن ها را در زیر می آوریم.

گرفتن ورودی:

نحوه گرفتن ورودی در پروژه به دو شکل گرافیکی و XML در متن پروژه تعریف شد. ورودی گرافیکی نمودار بخش بسیار زیادی از وقت پروژه را بدون اینکه قابلیت خاصی به پروژه اضافه کند گرفت. همچنین بهتر بود این رابط گرافیکی نه در زبان جاوا بلکه در صفحه وب پیاده سازی می شد که هم نتیجه زیباتری داشت و هم چالش کمتری داشت.

گرفتن ورودی XML چالش مهمی نداشت ولی بهتر بود که از قالب های دیگری مانند YAML استفاده می شد تا هم حجم فایل به خاطر تکرار دوباره تگ ها بالا نرود و هم خواندن آن راحت تر باشد.

پیاده سازی متد ها:

یکی از چالش ها زبان سی نداشتن اضافه بار توابع بود که قابلیت ارث بری و چندریختی متد ها را با دشواری روبه رو کرد.

خارج کردن توابع از ساختمان داده کلاس ها باعث شد تا قابلیت چندریختی پارامتری میسر نشود و بهتر بود که مانند هسته لینوکس که متد ها را در داخل کلاس ها ذخیره سازی می کند برنامه این روش را پیش می گرفت.

همگام کردن تحلیل گر لغوی:

تحلیل گر لغوی تنها بخش پیاده سازی شده پروژه در زبان پایتون بود. دلیل اینکار مناسب بودن تحلیلگر لغوی پیاده سازی شده در پایتون بود. روش ارتباطی تحلیل گر از طریق فایل بود. بهتر است روش ارتباطی از طریق فایل به روش اتصال کاربر و میزبان تغییر کند.

منابع و مراجع

- [1] Computer-aided software engineering. (Retrieved 2021, May 23). In *Wikipedia*.
https://en.wikipedia.org/wiki/Computer-aided_software_engineering
- [2] Leng, J., & Sharrock, W. (2011). *Handbook of Research on Computational Science and Engineering: Theory and Practice* (1st ed.). IGI Global.
- [3] Source-to-source compiler. (Retrieved 2021, September 15). In *Wikipedia*.
https://en.wikipedia.org/wiki/Source-to-source_compiler
- [4] Aaby, A. A. (1996). *Introduction to Programming Languages* (Retrieved 2021, October 12), from <http://www.worldcolleges.info/sites/default/files/aaby.pdf>
- [5] Mitchell, J. C. (2002). *Concepts in Programming Languages* (1st ed.). Cambridge University Press.
- [6] Inheritance (object-oriented programming). (Retrieved 2021, September 29). In *Wikipedia*. [https://en.wikipedia.org/wiki/Inheritance_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))
- [7] *The Four Polymorphisms in C++*. (Retrieved 2021, September). Catonmat.
<https://catonmat.net/cpp-polymorphism>
- [8] GeeksforGeeks. (Retrieved 2021, June 28). *Overriding in Java*.
<https://www.geeksforgeeks.org/overriding-in-java/>
- [9] Preprocessor. (Retrieved 2021, September 2). In *Wikipedia*.
<https://en.wikipedia.org/wiki/Preprocessor>
- [10] *Typeof (Using the GNU Compiler Collection (GCC))*. (Retrieved 2021, October 12). GCC, the GNU Compiler Collection.
<https://gcc.gnu.org/onlinedocs/gcc/Typeof.html>
- [11] *Other Builtins (Using the GNU Compiler Collection (GCC))*. (Retrieved 2021, October 12). GCC, the GNU Compiler Collection.
<https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>
- [12] *UML Distilled Third Edition* (3rd ed.). (2021). Addison-Wesley Professional.

[۱۳] م. رزازی، اصول طراحی کامپایلر، در حال چاپ

- [14] Douglass ,B. P. (2009). *UML for the C programming language*. [White Paper]. IBM
<https://silo.tips/download/uml-for-the-c-programming-language>
- [15] Astah. (Retrieved 2020, September 25). *UML2C Export Plug-In*.
<https://astah.net/product-plugins/uml2c-export/>
- [16] Cfront. (Retrieved 2020, October 1). In *Wikipedia*.
<https://en.wikipedia.org/wiki/Cfront>
- [17] Brown, N. (Retrieved 2011, June 1). *Object-oriented design patterns in the kernel, part 1*. LWN.Net. <https://lwn.net/Articles/444910/>
- [18] Brown, N. (Retrieved 2011, June 7). *Object-oriented design patterns in the kernel, part 2*. LWN.Net. <https://lwn.net/Articles/446317/>
- [19] Pedamkar, P. (Retrieved 2021, April 9). *Advantages of OOP*. EDUCBA.
<https://www.educba.com/advantages-of-oop/>