|  |
| --- |
| Архитектура на ArgoUML |
|
| *Проект по „Софтуерни архитектури“, 2016-17 г.* |
| Факултет по математика и информатика, Софийски университет |

|  |
| --- |
| Изготвили: |
| *80972*, *Кристиян Миленов Митов*, *kmitov94@gmail.com*  *Компютърни науки, четвърти курс, втори поток, пета група* |
| *80940*, *Николай Колев Георгиев*, *nikolaykgeorgiev@gmail.com*  *Компютърни науки, четвърти курс, втори поток, пета група* |
| *80981*, *Теодор Райчов Кирилов*, *sys146@gmail.com*  *Компютърни науки, четвърти курс, втори поток, шеста група* |
|  |

Ръководител: доц. Димитър Биров

2016 - 2017 г.

Съдържание

[1 Увод 6](#_Toc473423206)

[1.1 Тема на проекта 6](#_Toc473423207)

[1.2 Цели на проекта 7](#_Toc473423208)

[1.3 Резултати 7](#_Toc473423209)

[2 Описание 7](#_Toc473423210)

[3 Общ преглед на ArgoUML 9](#_Toc473423211)

[3.1 Отворен код, отворени стандарти 9](#_Toc473423212)

[3.2 Технологични иновации 9](#_Toc473423213)

[3.3 Платформата 10](#_Toc473423214)

[3.4 История на версиите 10](#_Toc473423215)

[3.5 Инсталиране и системни изисквания 10](#_Toc473423216)

[4 Изисквания към ArgoUML 10](#_Toc473423217)

[4.1 Официални изисквания пред ArgoUML 11](#_Toc473423218)

[4.1.1 Изисквания за потребителския интерфейс (Look And Feel) 12](#_Toc473423219)

[4.1.2 Изисквания за UML 14](#_Toc473423220)

[4.1.3 Изисквания за Java и JVM 15](#_Toc473423221)

[4.1.4 Изисквания за улесняване разработката на продукта 15](#_Toc473423222)

[4.2 Други функционални изисквания. Сценарии на употреба 16](#_Toc473423223)

[4.2.1 Създаване и редактиране на диаграми 16](#_Toc473423224)

[4.2.2 Поддръжка на различни визуализации 16](#_Toc473423225)

[4.2.3 Поддръжка на списъци с предстоящи задачи 17](#_Toc473423226)

[4.2.4 Грижа за качеството и коректността на моделите 18](#_Toc473423227)

[4.3 Нефункционални изисквания (качествени атрибути) 18](#_Toc473423228)

[4.3.1 Лекота за употреба (Usability) 19](#_Toc473423229)

[4.3.2 Възможност за разширяване на функционалността (Extensibility, Maintainability) 21](#_Toc473423230)

[4.3.3 Интерактивност и Производителност (Interactivity, Performance) 22](#_Toc473423231)

[4.3.4 Надеждност (Reliability) 24](#_Toc473423232)

[5 Функционалност, предоставена чрез потребителския интерфейс на ArgoUML 25](#_Toc473423233)

[5.1 Главен прозорец на ArgoUML 25](#_Toc473423234)

[5.1.1 Лента с инструменти 26](#_Toc473423235)

[5.1.2 Лента с менюта 26](#_Toc473423236)

[5.1.3 Раздел за разглеждане 27](#_Toc473423237)

[5.1.4 Редактор 28](#_Toc473423238)

[5.1.5 Раздел за предстоящи дейности 29](#_Toc473423239)

[5.1.6 Раздел с детайли 30](#_Toc473423240)

[5.2 Създаване на Клас диаграма 34](#_Toc473423241)

[5.2.1 Именуване на Проекта 34](#_Toc473423242)

[5.2.2 Именуване на Модела и Диаграмата 35](#_Toc473423243)

[5.2.3 Създаване на Клас и Именуването му 36](#_Toc473423244)

[5.2.4 Добавяне на Атрибути 37](#_Toc473423245)

[5.2.5 Създаване на операция с атрибутите 38](#_Toc473423246)

[5.2.6 Използване на Стилове 38](#_Toc473423247)

[5.2.7 Преместване, Преоразмеряване и Изобразяване на Асоциации 39](#_Toc473423248)

[5.2.8 Добавяне на Множественост 40](#_Toc473423249)

[5.2.9 Добавяне на Интерфейс 41](#_Toc473423250)

[5.2.10 Добавяне на реализация на Интерфейс 42](#_Toc473423251)

[5.2.11 Добавяне на генерализация 43](#_Toc473423252)

[6 Софтуерна архитектура на ArgoUML 44](#_Toc473423253)

[6.1 Компоненти на архитектурата 45](#_Toc473423254)

[6.2 Връзки между компонентите в ArgoUML 46](#_Toc473423255)

[6.3 Общ преглед на архитектурата 48](#_Toc473423256)

[6.4 Слой на системите от ниско ниво 50](#_Toc473423257)

[6.4.1 Подсистема за интернационализация (internationalization) 50](#_Toc473423258)

[6.4.2 Подсистема за менажиране на задачи 51](#_Toc473423259)

[6.4.3 Подсистема за конфигурация 52](#_Toc473423260)

[6.4.4 Подсистема за персистентност (persistence) 53](#_Toc473423261)

[6.4.5 Подсистема за logging 54](#_Toc473423262)

[6.4.6 Подсистема за модела 56](#_Toc473423263)

[6.4.7 Подсистема за UML профили 58](#_Toc473423264)

[6.5 Слой за преглед и контрол 60](#_Toc473423265)

[6.5.1 Подсистема за диаграми 61](#_Toc473423266)

[6.5.2 Подсистема за панели със свойства 62](#_Toc473423267)

[6.5.3 Подсистема за преглеждане на елементи (explorer) 64](#_Toc473423268)

[6.5.4 Подсистема за нотация 65](#_Toc473423269)

[6.5.5 Подсистема за графичен потребителски интерфейс (GUI) 66](#_Toc473423270)

[6.5.6 подсистема за зареждане на модули 67](#_Toc473423271)

[6.6 Слой от най-високо ниво – подсистема приложение (application) 68](#_Toc473423272)

[6.7 Подсистеми, които могат да се зареждат (loadable modules, plugins) 68](#_Toc473423273)

[6.7.1 Подсистеми за генериране и reverse engineering на код 69](#_Toc473423274)

[6.7.2 Подсистема за списъци с предстоящи задачи 70](#_Toc473423275)

[6.7.3 Подсистема за работа с OCL 71](#_Toc473423276)

[6.7.4 Подсистема за критици и помощни инструменти 71](#_Toc473423277)

[6.8 Външни подсистеми 73](#_Toc473423278)

[6.9 Deployment 73](#_Toc473423279)

[7 Оценка на архитектурата на ArgoUML 74](#_Toc473423280)

[7.1 Предимства 74](#_Toc473423281)

[7.1.1 Разделяне на архитектурата на подсистеми 74](#_Toc473423282)

[7.1.2 Използване на шаблона за дизайн Model-VIew-Controller(MVC) 75](#_Toc473423283)

[7.1.3 Поддръжка на динамично зареждащи се модули и плъгини 76](#_Toc473423284)

[7.1.4 Подобрение на качеството с помощта на останалите подсистеми 76](#_Toc473423285)

[7.2 Недостатъци 77](#_Toc473423286)

[8 Алтернативна архитектура 78](#_Toc473423287)

[8.1 Обща структура 78](#_Toc473423288)

[8.2 Изпълнение на модулите 79](#_Toc473423289)

[8.3 Обмен на данни 80](#_Toc473423290)

[8.4 Предаване на събития 80](#_Toc473423291)

[8.5 Плъгини 81](#_Toc473423292)

[8.6 Описание на новата архитектура, приложена върху ArgoUML 81](#_Toc473423293)

[8.6.1 Главен модул (Main Module) 82](#_Toc473423294)

[8.6.2 Модул за потребителски вход (User Input Module) 82](#_Toc473423295)

[8.6.3 Модул за менюта (Menu Module) 82](#_Toc473423296)

[8.6.4 Модул за управление на диаграми (Diagram Manager Module) 83](#_Toc473423297)

[8.6.5 Модул за модел на диаграмите (Diagram Model Module) 83](#_Toc473423298)

[8.6.6 Модул за изглед на диаграмите (Diagram View Module) 83](#_Toc473423299)

[8.6.7 Модул за генериране на програмен код (Generate Source Module) 84](#_Toc473423300)

[8.6.8 Модул за достъп до файловата система (IO Module) 84](#_Toc473423301)

[8.6.9 Други модули и плъгини 84](#_Toc473423302)

[8.7 Предимства на новата архитектура 84](#_Toc473423303)

[8.8 Сравнение със старата архитектура 85](#_Toc473423304)

[9 Заключение и оценка 86](#_Toc473423305)

[Библиография 88](#_Toc473423306)

1. Увод

През 90-те години на 20-ти век обектно-ориентираните технологии напълно завършват прехода си от експериментални техники към общоприета софтуерна парадигма. Това е съпроводено с трудности, породени от нуждата да се промени мисловния процес на системните дизайнери, разработчици и другите участници в цикъла на разработка на софтуер. С появата на поддръжка за обектно-ориентирано програмиране в редица езици като С++, Ада и други обаче то бързо набира популярност, защото позволява софтуерът да се изгражда като симулация на реалния свят и позволява повече възможности за дизайн и тестване на системи преди дори да бъдат имплементирани. Освен това за пръв път моделирането може да става на по-високо ниво, което позволява и изграждането на по-сложни системи от смятаното за постижимо дотогава.

Поради голямото разнообразие от методологии, които обектно-ориентираното програмиране предлага, бързо става ясна нуждата от унифицирано средство за предаване на обектно-ориентирани концепции между отделните участници в процеса на дизайн. В отговор на тази нужда възниква Unified Modelling Language (UML), но неговата поява поражда и нужда от нови инструменти, които да позволяват автоматизация и улесняване на процеса на дизайн с помощта на графичен потребителски интернет. Сред тях е ArgoUML – среда за анализ и дизайн на обектно-ориентирани софтуерни системи с помощта на UML.

* 1. Тема на проекта

Обект на разглеждане в този проект е ArgoUML като помощен инструмент за работа с UML. Той е базиран на спецификацията на UML 1.4, но предоставя и поддръжка за Object Contraint Language (OCL) и XMI (XML Model Interchange format), както и редица други отворени стандарти. Неговите създатели си поставят задача да предоставят помощни средства, които да отговарят на реалните нужди на дизайнерите на софтуер. Ние ще го разгледаме като типичен образец на една такава система, ще извлечем неговата архитектура, ще анализираме силните и слабите й страни и предложим алтернативна архитектура, която да се справи с някои от недостатъците.

* 1. Цели на проекта

Основна цел на проекта е запознаване с процеса на създаване, анализ и оценка на софтуерни архитектури, като се съсредоточим върху конкретен проект – създаване на приложение за работа с UML диаграми. С оглед на това дефинираме следните подцели: да извлечем изискванията към такъв тип продукт, съсредоточавайки вниманието си особено върху нефункционалните изисквания; да извлечем и анализираме архитектурата на приложението към момента заедно със силните и слабите й страни; да предложим алтернативна архитектура, която да се справи с някои от слабостите; да направим цялостна оценка и финални заключения.

* 1. Резултати

След детайлно разглеждане на продукта ArgoUML се запознахме с неговата архитектура и установихме връзка между извлечените от нас нефункционални изисквания за такъв тип приложения и реално взети архитектурни решения в продукта. Освен това успяхме да установим някои слаби места във въпросната архитектура и да предложим алтернативна такава, която да разреши този проблем.

1. Описание

С оглед на това, че ще разглеждаме архитектурата на ArgoUML, следва първо да се запознаем с някои базови характеристики на продукта. Разделът “Общ преглед” съдържа описание на особеностите на инструмента, основните му предимства пред други подобни продукти, история на създаването му и предишни версии. В този раздел са представени най-общо възможностите, които той предоставя на потребителя.

В раздела “Изисквания към ArgoUML” ще представим основните софтуерни изисквания, които възникват при разработката на система като ArgoUML – функционални и нефункционални. Част от тях са извлечени чрез анализ на инструмента и проблемите, които той цели да разреши, а останалите са посочени директно от създателите на системата в съпътстващата документация. Ще бъдат представени някои типични сценарии на употреба за ArgoUML със съпътстващи use-case диаграми. Особено внимание ще бъде обърнато на нефункционалните изисквания, характеризирани по шестстъпковия модел.

В раздела “Функционалност, предоставена чрез потребителския интерфейс на ArgoUML” ще покажем обстойно възможностите, които ArgoUML предоставя на потребителите чрез графичния си интерфейс. Това включва основните екрани и менюта, опции и начини за употреба. По този начин ще бъде изяснени функционалните изисквания, във връзка с които е възникнала оригиналната и на които трябва да отговаря и нашата архитектура.

В раздела “Софтуерна архитектура на ArgoUML” ще разгледаме ще разгледаме избраната от разработчиците на ArgoUML архитектура. Ще изпозваме диаграми с компоненти и конектори, модулни диаграми и deployment диаграма. Ще обърнем внимание на всеки компонент поотделно, връзките му с останалите компоненти, функционалностите, които предоставя, и някои основни детайли от имплементацията му.

В раздела “Оценка на архитектурата на ArgoUML” ще преминем към анализ на разгледаната архитектура. Ще откроим връзката между взетите в нея решения и функционалните изисквания. Ще отбележим предимства и недостатътци.

В раздела “Алтернативна архитектура” ще представим предложена от нас архитектура, която ще се опита да се справи с описаните преди това недостатъци. Ще мотивираме решенията си в контекста на качествените атрибути, които разглеждаме.

1. Общ преглед на ArgoUML

ArgoUML e замислен като едновременно помощно средство и среда за дизайн и анализ на обектно-ориентирани системи. Той е подобен на много инструменти за компютърно-подпомогнато моделиране, но се различава от тях по няколко важни характеристики.

* 1. Отворен код, отворени стандарти

ArgoUML e един от първите подобни продукти, който се разпространява безплатно. Освен това е проект с отворен код, което гарантира, че ново поколение софтуерни дизайнери и изследователи могат да го използват като утвърдена база за надграждане в хода на еволюцията на технологиите за моделиране. Тясно свързан с това е и фактът, че ArgoUML поддържа отворени стандарти – не само UML, но и XMI, OCL, SVG и други. Напълно се поддържа UML 1.4, както и генериране на код по статичните класови диаграми.

* 1. Технологични иновации

От технологична гледна точка ArgoUML се откроява с това, че е изграден въз основа на изследвания в сферата на психологията, за да предостави иновативни средства за повишаване на продуктивността, отговарящи на нуждата на софтуерните дизайнери и архитекти. Базира се на три основни принципа: разсъждение в действие (дизайнерите не си представят цялата системата, а създават частичен дизайн, оценяват го, подобряват го и итерират), опортюнистичен дизайн (дизайнерите правят планове, но изпълняват отделните стъпки от тях не подред, а според това коя що коства най-малко усилие първо) и разбиране на проблема (дизайнерите трябва да свържат умствения си модел с формалния). За целта се представят средства за множество различни визуализации от различни гледни точки, сравняване на текущия дизайн с модели за най-добри практики (т.нар. дизайн критици), удобна интеграция на списък с предстоящи дейности и списъци с отметки, за да има потребителят винаги поглед върху свършеното до момента и предстоящата работа.

* 1. Платформата

С цел да бъде широкодостъпен той е изграден изцяло върху Java платформата, като 100% чисто джава приложение. Това му позволява да работи на всяка платформа и операционна система, където е налична Java виртуална машина. Допълнителна причина за избора на Java е широката й популярност и продуктивност, което, съчетано с популярността на UML, превръща ArgoUML в привлекателно средство за разработка, използвано не само в индустрията, но и като поле за тестване на иновативни подходи от пионерите в софтуерното инженерство.

* 1. История на версиите

Продуктът е разработен от екип от Университета на Калифорния, ръководeн от Jason Robbins. Първата версия става публично достъпно през 1998 година до средата на 2001 има повече от 100000 сваляния – знак за популярност в образователните среди и индустрията. Оттогава се разработва от по-широк кръг хора, но страда от недостиг на хора, поради което последната достъпна версия е 0.34 от 15 декември, 2011 година.

* 1. Инсталиране и системни изисквания

ArgoUML има сравнително ниски системни изисквания – най-вече да има инсталирана Java виртуална машина и 15 мегабайта дисково пространство. За Уиндоус има предоставен инсталатор, може да се свалят директно jar файловете или да се отвори директно през браузъра чрез Java Web Start. За всяка версия е предоставен и пълен достъп до програмния код. Разпространява се под Eclipse Public License.

1. Изисквания към ArgoUML

Като всеки софтуерен продукт, ArgoUML е плод на дълъг процес на разработка. Първата фаза на всеки такъв проект е анализ и извличане на изискванията, на които трябва да отговаря крайният продукт. Запознаването с тези изисквания е важна час от процеса на анализ на архитектурата на продукта, защото много от имплементираните функционалности и взетите решения обикновено са директно следствие от някое от изискванията. Затова ще представим основните функционални изисквания, на които отговаря продуктът, официалните изисквания, документирани от създателите на системата, и нефункционалните изисквания, извлечени от нас, за система като ArgoUML. Те ще служат за база за оценка както на архитектурата на продукта, така и на предложената от нас алтернативна архитектура, която да коригира някои от слабостите на оригиналната.

* 1. Официални изисквания пред ArgoUML

В официалното wiki на ArgoUML има специална страница [ARGRQ], в която разработчиците са дефинирали изискванията, към които трябва да се приддържа приложението. Включени са както вече имплементирана функционалност, така и изисквания за бъдещи разширения на продукта, като е възможно решението още да не е ясно, но ясно е дефинирана крайната цел. Това е важно поради две причини: на първо място, ArgoUML е отворен проект и всеки може да допринася за развитието на продукта, но това може да породи хаос и неконсистентност, особено на архитектурно ниво. Ако обаче има ясни и публични изисквания, към които да се придържат всички, това ще доведе до унифициран дизайн, предвидимост на поведението на продукта и може да предотврати по-нататъшни проблеми. На второ място публичното представяне на изискванията помага бързо да се открият проблеми в кода – всяко нещо, което не отговаря на очакванията, породени от тях, може да се регистрира като бъг и това се насърчава от разработчиците на продукта.

Следва да представим основните официални изисквания, посочени на страницата. Както е прието в процеса на дизайн на софтуер, те са номерирани с REQ1, REQ2 и тн, имат ревизии, обозначавани с REVa, REVb, REVc и всяко е съпроводено с описание, причина да бъде поставено като важно и стейкхолдър – за кои човек или група е важно това изискване.

* + 1. **Изисквания за потребителския интерфейс (Look And Feel)**

Едни от най-важните изисквания са тези, свързани с това, как изглежда и се държи продукта. Това са предимно функционални изисквания, свързани с функционалности, които трябва да се имплементират, но има и някои от тях, които са свързани по-скоро с поддържането на определени свойства на продукта като консистентност, надежност, удобство за използване и т.н.

* + - 1. **Изискване за консистентен ъпдейт REQ1 REVa**

Това изискване гласи, че когато множество компоненти показват същия елемент от модела, те трябва да бъдат обновявани консистентно из цялото приложение. Причината за него е, че няма как да се знае къде гледа потребителят в даден момент, докато работи с продукта, и не бива той да има причина да се обърка, ако два различни изгледа към един и същи обект представят различни неща. Затова и стейкхолдърът, за който е важно това изискване, е именно потребителят.

* + - 1. **Изискване за ъпдейт при всяко натискане на клавиш и клик REQ2 REVb**

Всички изгледи на модела, които могат да го редактират, трябва да се обновяват при всеки клик или натискане на клавиш. Идеята е, че ако потребителят обнови част от модела, незабавна обратна връзка в другите му части може да го улеснят в по-нататъшни промени. Стейкхолдър отново е потребителят.

* + - 1. **Изискване за редактиране на полета, изискващи валидация REQ12 REVa**

Текстови полета, които изискват валидация, не трябва да могат да се редактират директно през изглед. Причината е, че има вероятност полето да е в невалидно състояние във всеки момент по време на самото му редактиране, и моделът не може да се ъпдейтне, докато полето не бъде завършено във валидно състояние или опитът за редакция не бъде отхвърлен от валидацията. Стейкхолдър са потребителите на ArgoUML, но има отворен въпрос дали това е правилният избор на стейкхолдър.

* + - 1. **Изискване за диалоговите прозорци REQ13 REVa**

Когато потребителят попълва информация в диалогов прозорец, моделът не се обновява, докато диалоговият прозорец не бъде приет от потребителя с попълнени полета. Въвежда се това изискване, защото в повечето потребителски интерфейси диалогът представя моделът така, както е изглеждал в момента на създаването на прозореца, и обновяване се извършва само след затворянето му с приемането от потребителя на промените в него. Това е познато за потребителя поведение. Именно той е стейкхолдър и тук.

* + - 1. **Изискване за визулна обратна връзка за валидност REQ14 REVa**

Потребителят трябва да получава визуална обратна връзка по време на процеса на редактиране на текстови UML, за да може да разбере дали написаното от него е валиден синтаксис. Целта е улеснение – спазването на синтактични правила е сложен процес и в писането на код компилаторите помагат в откриването на проблеми със синтаксиса, но това не е налице при текстовите полета – вместо това се прави валидация още по време на редактиране, за да може потребителят да получи помощ възможно най-бързо, за да спази правилата на синтаксиса. Стейкхолдър отново е потребителят на ArgoUML.

* + - 1. **Изискване за визуализация на грешки REQ3 REVa**

Не трябва да има индикация за изключение или грешка на екрана, ако потребителят просто е сбъркал синтаксиса на UML, понеже такава индикация обикновено е знак за грешка в приложението и се възприема като дефект. Тъй като грешките на потребителя са често нещо, ако генерират такъв проблем, може той да загуби интерес към ArgoUML, защото от негова гледна точка продуктът не работи коректно. Стейкхолдър е потребителят на ArgoUML.

* + - 1. **Изискване за помощна информация REQ4 REVa**

Всички текстови полета трябва да предоставят помощна информация, при това подходяща в зависимост от контекста на употреба. По-конкретно, при плъзгане на мишката върху полето би трябвало да се появи информация, обясняваща типа данни и формат, очаквани от полето, освен когато има етикет към полето и форматът е очевиден. Освен това при натискане на F1 или отваряне на помощна информация от менюто трябва да се появи прозорец, обясняващ данните и формата, очаквани от текущото поле, което е на фокус, като то запазва фокуса при интеракция с отворения прозорец (влачене, скролиране, затваряне). Причината за това изискване е, че във всяко по-сложно приложение има много текстови полета и потребителят трябва да може да се възползва от такава информация, за да ги използва правилно. Затова и стейкхолдър е именно потребителят.

* + 1. **Изисквания за UML**
       1. **Коректност на имплементацията REQ5 REVb**

ArgoUML трябва да е коректна имплементация на UML 1.4, като има и план за придвижване към UML 2.0. Това е важно изискване, свързано с основното предназначение на продукта да е средство, което да подпомага работата на дизайнерите с UML модели, които трябва да могат да бъдат използвани в друг инструмент. За целта имплементацията трябва да е вярна, иначе няма да има съвместимост с други инструменти или пък ще предизвиква объркване на потребителя поради различни резултати. Стейкхолдър са потребителите на ArgoUML

* + - 1. **Пълнота на имплементацията REQ6 REVb**

ArgoUML трябва да имплементира всичко в UML 1.4 модела. Така както и да използва UML потребителя, ArgoUML винаги ще бъде добър избор за потребителя, при това работещ коректно. Стейкхолдър са потребителите на ArgoUML.

* + - 1. **Налагане на правила за правилно оформяне по UML стандарта REQ15 REVa**

ArgoUML предупреждава за нарушаване на правилата за правилно оформление чрез критиците, но не ги налага, с някои изключения, като изключенията трябва да са документирани и продуктът да се справя с тях консистентно. Дори когато се налагат правила, ArgoUML трябва да може да работи с модели, които ги нарушават. Целта е максимална гъвкавост на приложението, а стейкхолдър е потребителят.

* + 1. **Изисквания за Java и JVM**
       1. **Поддръжка на съвместими JRE REQ7 REVb**

ArgoUML трябва да поддържа всяка JRE, съвместима със спецификация на Sun, която не е започнала Sun End of Life (EOL) процес. Причината е, че JRE със съпътстващите библиотеки непрекъснато се подобряват, за да включват нова функционалност и идеи, от които разработчиците на ArgoUML биха искали да се възползват. Стейкхолдър са разработчиците на ArgoUML.

* + - 1. **Лесно сваляне от интернет и стартиране REQ8 REVa**

Трябва да е възможно да се инсталира ArgoUML както през интернет, така и локално на машината, и освен това трябва да е възможно използването му без интернет връзка. Това е така, защото ArgoUML е приложение за редактиране на UML модели, няма нужда от връзка с мрежата, за да осъществи тази си задача. Стейкхолдър са потребителите на ArgoUML.

* + - 1. **Изход на командния прозорец REQ9 REVa**

Информация, свързване с документиране на хода на програмата и проследяване на изпълнението, не трябва да бъде визуализирана на конзолата или отпечатвана във файл освен ако това не е изрично включено от потребителя. Причината е, че ArgoUML цели да позволява работа с UML модели, всяка информация, записвана другаде, освен в посочените от потребителя файлове, може да бъде неразбираема за него и да бъде възприемана като ненужно генерирана и безсмислена. Стейкхолдър са потребителите на ArgoUML.

* + 1. **Изисквания за улесняване разработката на продукта**
       1. **Документиране на хода на действие на програмата REQ10 REVa**

Кодът трябва да бъде написан така, че да е възможно да се записва информация за всички по-важни моменти от работата на приложението с цел подпомагане на лесно откриване на проблеми от разработчиците на ArgoUML. Целта е както самите разработчици по-лесно и бързо да отстраняват бъгове при разработката, така и да могат да се ориентират и своевременно да окажат съдействие на потребител, съобщаващ за проблем. Стейкхолдър са именно разработчиците на ArgoUML.

## Други функционални изисквания. Сценарии на употреба

Макар да не са официално документирани на страницата на проекта, съществуват и редица други, предимно функционални изисквания, които трябва да бъдат поставени, за да се постигне функционалност, съизмерима с тази на ArgoUML. Ще представим основните такива функционални изисквания, т.е. каква поддържа съвременната архитектура и трябва да бъде съхранено при каквито и да е нейни модификации.

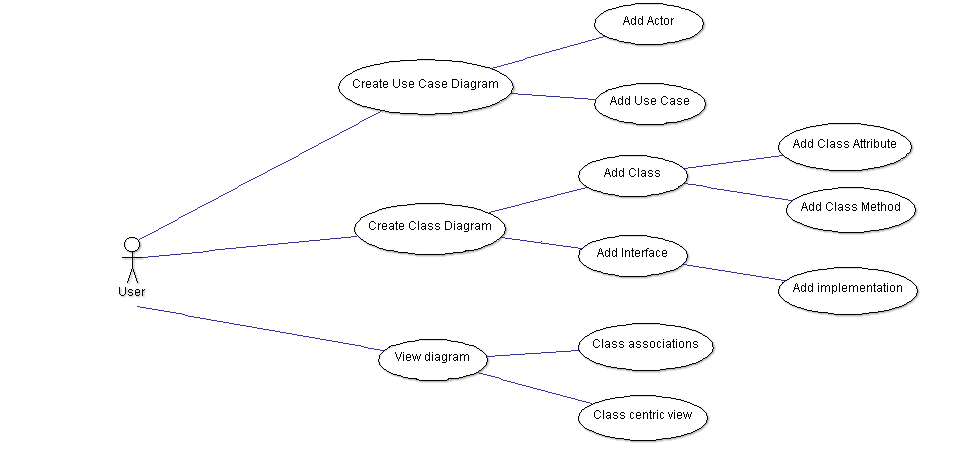
### Създаване и редактиране на диаграми

Като продукт за работа с UML основната функционалност, която трябва да поддържа, е създаване и редактиране на UML диаграми. Това включва поддръжка на основните типове диаграми: диаграми за случаите на употреба, класови диаграми, диаграми на последователностите, диаграми на активности, диаграми на състоянието, компонентни диаграми. Потребителят трябва да може да добавя атрибути и асоциации, интерфейси и техни реализации, да анотира диаграмите с бележки, да променя цвета, размера и други характеристики на компонентите в модела. Целта е ArgoUML да бъде гъвкав и предпочитан избор благодарение на широката функционалност, която предоставя. Стейкхолдъри за това изискване са потребителите.

### Поддръжка на различни визуализации

ArgoUML трябва да поддържа различни визуализации на разглеждания модел – например групиране спрямо асоциациите между елементите на модела, гледни точки към различни негови елементи и други. Причината за това изискване се корени в една от особеностите на продукта – вдъхновен от изследвания за човешката психология, които показват, че дизайнерите имат тенденцията да итерират, променят последователно отделни части от модела и след това, след като са придобили нова перспектива, да правят нови промени, ArgoUML следва да улесни това, доколкото е възможно. Стейкхолдър и за това изискване са потребителите на продукта.

Следната диаграма на случай на употреба представя едни от най-честите сценарии, свързани с горните изисквания – създаване на диаграми и разглеждане на вече съществуващи модели в различни перспективи.



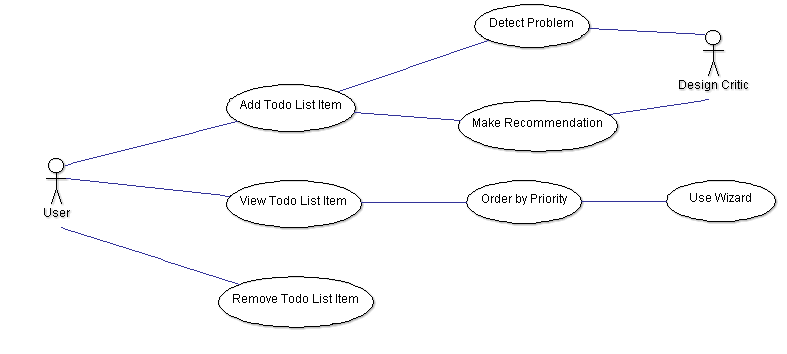
**Фиг. 1.** *Диаграма за случай на употреба – създаване и разглеждане на модели*

### Поддръжка на списъци с предстоящи задачи

ArgoUML трябва да прави възможно поддържане на списъци с предстоящи задачи. Потребителите трябва да могат да добавят, следят прогреса на задачите и да ги маркират като успешно завършили. Освен това трябва да могат да ги филтрират или сортират по различни критерии, например приоритет. Ценна допълнителна функционалност е способността част от предстоящите задачи да са автоматично генерирани от софтуера предложения, базирани на текущите най-добри практики за дизайн. С това могат да бъдат привлечени повече потребители, понеже в сложните проекти често някои предстоящи задачи се пренебрегват или забравят, а дори и да не се случи това, невинаги е лесно да се подредят по приоритет, да се прецени кои са важни и кои не и т.н. Улеснявайки този процес, ArgoUML има важно предимство пред конкуретните продукти. Стейкхолдър за това изискване са потребителите.

### Грижа за качеството и коректността на моделите

Продуктът трябва да следи за качеството и коректността на генерираните чрез него модели. Това става чрез поддръжка за проверка на синтаксиса и генериране на предупреждения при нарушаване на правилата за добро оформление, дефинирани в стандарта. Освен това трябва да се даде възможност на потребителя да дефинира свои собствени ограничения, продиктувани от домейна, чрез OCL и те да бъдат налагани от самия софтуер. Не на последно място ценна функционалност е даването на предложения за промени въз основа на предварително известен набор от добри практики. Всичко това цели да улесни максимално развитието на проекта като цяло чрез отстраняване на грешки още в зародиша им и насочване на потребителя към потенциално по-добри решения. Стейкхолдър отново са именно потребителите.



**Фиг. 2.** *Диаграма за сценарий – работа със списък с предстоящи задачи*

## Нефункционални изисквания (качествени атрибути)

От изключителна важност за всяка съвременна софтуерна система освен функционалните са и нефункционалните изисквания. Докато функционалните изисквания лесно се свързват с конкретното поведение на приложението и в този смисъл по-лесно се проверява тяхното спазване, нефункционалните изисквания налагат по-дълбоко изследване, внимателно обмисляне на начина, по който ще оценяваме тяхното спазване, и също толкова внимателно обмисляне на архитектурата на бъдещия ни продукт. Последното тук е от особено значение, защото, докато останалите изисквания обикновено се свързват с отделен компонент на архитектурата, качествените атрибути трябва да залегнат в нея като цяло, във всички подсистеми, и ако не внимаваме, лесно можем да стигнем до грешен дизайн, който после ще е много трудно и скъпо да рефакторираме. Затова при характеризирането на архитектурата на ArgoUML важна предварителна стъпка е уточняването на нефункционалните изисквания. Вземайки предвид спецификата на разглеждания от нас софтуер, който представлява десктоп приложение, което не е предвидено да оперира в разпределена страна, се открояват следните нефункционални изисквания, които според нас трябва да заемат приоритетно място при изграждането на архитектурата на приложението. Като софтуер за създаване на различни видове диаграми ArgoUML трябва да предлага на потребителите лекота за употреба, надеждност, интерактивност и възможност за лесно разширяване на функционалността – в случаи, когато потребителят има нужда от неимплементирана опция или желае да пригоди приложението към своя работен процес. Какво разбираме под тези четири атрибута ще обясним по-подробно, като използваме шест-стъпковия модел за харектеризация на качествени атрибути.

### Лекота за употреба (Usability)

На първо място за всяка система, предвидена за употреба от обикновения потребител, е нейната използваемост. Този качествен атрибут е мярка за това, колко е лесна за използване системата, дали е интуитивна за потребителя, дали е налична локализация и глобализация, остава ли доволен потребителят след използването й. Нека се опитаме да конкретизираме това, използвайки шестъпковия модел.

**Възможни източници**: Когато говорим за лекота на употреба, това е крайният потребител.

**Възможни стимули**: Възможни стимули са желание да се използва системата с цел постигане на резултат за максимално кратко време; желание да се разучи системата; желание да се модифицира системата за собствени цели; желание да се конфигурира системата; желание да се допуснат минимален брой грешки при използването й за създаването на модел.

**Възможни среди**: Действието може да се развива и по време на самата работа на приложението, и по време на конфигурацията му за постигане на определена цел.

**Възможни стимулирани артефакти**: Потребителският интерфейс или логиката на програмата или системите за помощ за потребителя.

**Възможни отговори**: Предоставяне на желаната функционалност, предугаждане на нуждите на потребителя и подпомагане със съвети.

**Възможни мерки за отговора**: Време, необходимо за завършване на задачата; време, необходимо за разучаване; брой или съотношение на изпуснати грешки; до колко доволен е останал потребителят.

Следва да разгледаме и конкретен сценарий:

**Източник**: Потребителят.

**Стимул:** Желание да разучи и създаде диаграма, каквато преди не е правил, като се стреми крайният модел да отговаря възможно най-добре на съвременните приети практики.

**Стимулиран артефакт**: Подсистемата за работа с диаграми, подсистемата за модела и подсистемата за потребителски интерфейс, потенциално подсистемата на дизайн критиците.

**Среда**: По време на работа на програмата.

**Отговор**: Предоставяне на желаната функционалност заедно със съвети за по-добър стил на диаграмата, предоставени от дизайн критиците.

**Оценка за отговора**: Време за успешно разучаване на диаграмата и постигане на резултат – желателно е да е за по-малко от час.

### Възможност за разширяване на функционалността (Extensibility, Maintainability)

Както знаем, съвременните софтуерни архитектури са достигнали високи степени на сложност и поради тази причина все по-често се използват най-различни визуализации за улесняване на разбираемостта. В резултат от това често се налага системи като ArgoUML да бъдат разширявани, за да поддържат нова семантика в старите или изцяло нови типове диаграми. Във връзка с това важен е качественият атрибут maintainability – способността на системата да бъде променяна с поне известна степен на лекота, за да се изпълнят нововъзникнали изисквания, без това да внесе твърде много сложност или да засегне твърде много съществуващите подсистеми. Конкретно в нашия случай това е наистина важно изискване за програмата, за да бъде полезна в по-големи проекти. Архитектурата трябва да позволява лесно разширяване на функционалността независимо от това, коя част от програмата желаем да променим. Да има възможност за промени по потребителския интерфейс. Да може лесно да се променя изгледа на елементите в диаграмите. Да може да се добавят алгоритми, които да обхождат данните за диаграмата и да ги променят – например когато потребителя има много на брой диаграми и иска да автоматизира процеса по изменението им.

**Възможни източници**: Може да бъде както обикновеният краен потребител, желаещ да адаптира програмата за свои нужди, така и разработчикът, желаещ да вкара изцяло нова функционалност, или просто по-напреднал потребител, който иска да автоматизира задачите си.

**Възможни стимули**: Възможни стимули са желание да се премахне, добави или промени функционалност или използвана за имплементацията й технология.

**Възможни среди**: Действието може да се развива и по време на самата работа на приложението, по време на конфигурацията му за постигане на определена цел, по време на разработката му, дори на фаза дизайн.

**Възможни стимулирани артефакти**: Системата като цяло, или само потребителския интерфейс, или платформата, върху която се разработва

**Възможни отговори**: Лесно се прави, тества и пуска в производство модифицираният вариант или това става, но трудно, или се оказва невъзможно въобще

**Възможни мерки за отговора**: Колко струва промяната, било то измерено в пари, време, усилие, сложност или друга метрика; също може да ни интересува дали сме създали дефекти в системата и ако да, какви и колко, сериозни ли са.

Следва да разгледаме и конкретен сценарий:

**Източник**: Потребителят.

**Стимул:** Желание да добави поддръжка за нов тип диаграма.

**Среда**: По време на разработка.

**Стимулиран артефакт**: Подсистемата за работа с диаграми и подсистемата на модела.

**Отговор**: Промяната е сравнително лесна, необходими са промени само по някои от подсистемите.

**Оценка за отговора**: Време за имплементиране, тестване и успешно интегриране на новия тип диаграма.

### Интерактивност и Производителност (Interactivity, Performance)

В съвременния свят все по-важно става използваните компютърни системи да реагират бързо на потребителския вход. Мярка да способността на системата да реагира в рамките на определено време е качественият атрибут интерактивност или производителност. В нашия случай потребителят често е притиснат от срокове и му предстои сложна задача, затова програмата трябва да реагира възможно най-бързо на входа му. Потребителският интерфейс не трябва да се забавя в случаи, когато са създадени много на брой диаграми. Програмата трябва да остава интерактивна дори при наличието на много голяма и сложна диаграма.

**Възможни източници**: Най-често това е потребителят, но може и да бъде друг компонент на системата, който в даден момент има нужда да инициира изпълнението на някакви задачи.

**Възможни стимули**: Възможни стимули са периодични, спорадични или случайно настъпващи събития в системата.

**Възможни среди**: Действието се развива по време на работа на приложението, но може да е в нормална обстановка, при високо натоварване (т.нар. overload режим) или при ограничени ресурси.

**Възможни стимулирани артефакти**: Системата като цяло или отделни компоненти.

**Възможни отговори**: Грешката се предотвратява или само се нотифицира потребителя или грешката се записва, но програмата прекратява работата си, или продължава нормалното функциониране на системата.

**Възможни мерки за отговора**: време за възстановяване след проблем или вероятност да се възстанови от него.

Следва да разгледаме и конкретен сценарий:

**Източник**: потребителят

**Стимул:** Желание да редактира сложна диаграма с множество отворени изгледи.

**Среда**: по време на работа, в нормален режим

**Стимулиран артефакт**: системата като цяло

**Отговор**: Промените се осъществяват, макар и с малко забавяне

**Оценка за отговора**: Латентност на операцията – за интерактивност трябва всяка модификация да се отразява в рамките на секунда на диаграмата (тук това е по-подходяща мярка, тъй като за ArgoUML потребителят обикновено е един и броят на едновременните операции няма как да е голям).

### Надеждност (Reliability)

Не на последно място за потребителя е важно системите, които използва, да са надеждни. Мярка да способността на системата да остане функционираща, и то правилно, за продължителен период от време, се нарича надеждност. Най-общо казано надеждността на програмата се изразява в това, да се минимизират непредвидените спирания на софтуера при поява на грешка. Ако проблемът е възникнал в част от програмата, която не се грижи за съхранението на данните за диаграмите, софтуерната архитектура на ArgoUML трябва да предоставя удобен механизъм за предпазване надиаграмата. В случаи, когато се работи по няколко диаграми едновременно и има незаписани промени, възникването на грешка, пряко свързана с една диаграма, не трябва да поврежда останалите. Такива възможности биха направили софтуера използваем дори в enterprise среди.

**Възможни източници**: Най-често това е потребителят, но може и да бъде друг компонент на системата, който в даден момент среща някаква грешка, или дори друг елемент от компютърната система, например хардуерът.

**Възможни стимули**: Възможни стимули са crash, потребителска грешка, системна грешка.

**Възможни среди**: Действието се развива по време на работа на приложението, но може да е в нормална обстановка, при високо натоварване (т.нар. overload режим) или при ограничени ресурси.

**Възможни стимулирани артефакти**: Системата като цяло или отделни компоненти или само потребителският интерфейс.

**Възможни отговори**: Обработват се събитията без забавяне или със забавяне или коренно се променя необходимото време за завършване на дадена задача.

**Възможни мерки за отговора**: най-често срещани мерки са латентността – колко време отнема завършването на операция, и throughput-ът – колко операции могат да се извършват едновременно.

Следва да разгледаме и конкретен сценарий:

**Източник**: Хардуерът.

**Стимул:** Възниква проблем с хардуера и системата се рестартира.

**Среда**: По време на работа, но в авариен режим за системата

**Стимулиран артефакт**: Системата като цяло, най-вече подсистемата за персистентност.

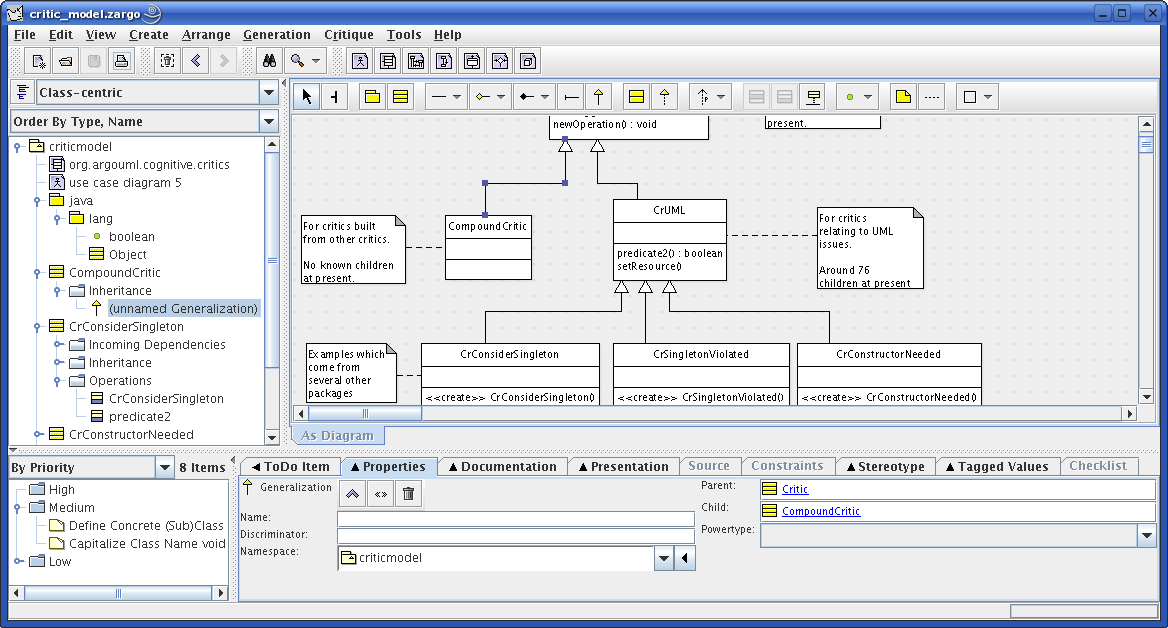
**Отговор**: Процесът умира, но промените по диаграмите се запазват.

**Оценка за отговора**: Каква част от промените са загубени – добре е да не е повече от последните 5-10 минути, за да може потребителят бързо за възстанови работата си.

# Функционалност, предоставена чрез потребителския интерфейс на ArgoUML

## Главен прозорец на ArgoUML

Главния прозорец е първото нещо, което се вижда, когато се отвори ArgoUML (фиг. 3). Този прозорец има лента с инструменти (фиг. 4), лента с менюта(фиг. 5) и 4 главни раздела: раздел за разглеждане(фиг. 6), редактор (фиг. 7), раздел за предстоящи дейности (фиг. 8) и детайли (фиг. 9).

**Фиг. 3**. *Главния прозорец на ArgoUML*

### Лента с инструменти

Лентата с инструменти (фиг. 4) се състои от икони/инструменти за най-използваните функции от менютата: операции с файлове, навигация, търсене и оразмеряване, и създаване на диаграми.

**Фиг. 4.** *Лента с инструменти*

### Лента с менюта

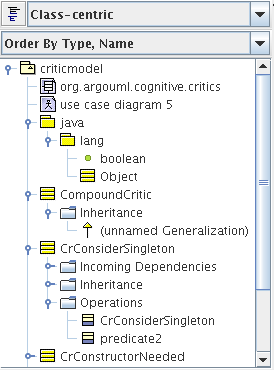
Лентата с менюта се състои от менюта за Файл, Редактиране, Изглед, Създаване на диаграми, Подредба, Генериране на код, Критика, Инструменти и Помощ.

**Фиг. 5**. *Лента с менюта*

### Раздел за разглеждане

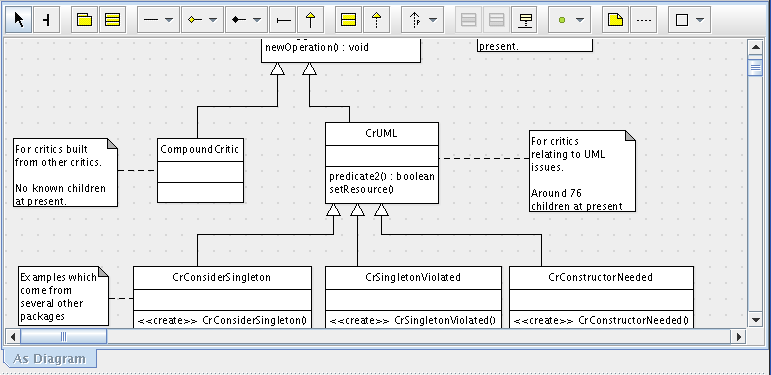
Панела за разглеждане (фиг. 6) показва съдържанието на дизайна. Този елемент би трябвало да е познат на всеки, който е ползвал Windows Explorer или друг файлов мениджър. Едно кликане върху обект, селектира дадения обект в редактора и показва неговите детайли в панела за детайли. Сортирането на елементите в дървовидната структура може да е азбучно, или (по подразбиране) по тип.

Уникално за ArgoUML са различните перспективи. Менюто в горната част на панела за разглеждане избира текущата перспектива. Всяка перспектива показва йерархичен изглед на дизайна, като подчертава дадени свойства и скрива други.Например, една перспектива прави наследяването ясно, докато друга се фокусира върху преходите между състояния. Няколко перспективи да предоставени с ArgoUML, като може да се дефинират допълнителни.

**Фиг. 6.** *Раздел за разглеждане*

### Редактор

Редактора (фиг. 7) е главния раздел за работа. Използва се за да се редактират диаграми. В горната си част има панел, който съдържа всички елементи (ModelElement), които могат да се нарисуват на текущия тип диаграма. Кликане върху избрания елемент селектира типа на елемента, който ще се появи в зоната за диаграма с просто кликане. Освен това, панелът съдържа общи инструменти за рисуване, за декорания на диаграмите с линии, текст и фигури, които не са част от UML модела.

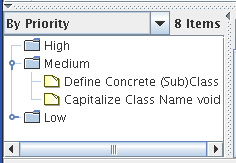
**Фиг. 7**. *Редактор*

### Раздел за предстоящи дейности

Раздел за предстоящи дейности (фиг. 8) помага на дизайнерите да помнят какво предстои да се направи. Елементите на този панел може да са лични напомняния, въведени от дизайнера, но по-често са генерирани от дизайнерската критика. Дизайнерските критики постоянно анализират дизайна, търсейки незавършени или проблемни зони. Когато потенциален проблем е намерен, критикът генерира „предстояща дейност“ и я добавя към списъка в панела. Когато се разреши проблема, съответният елемент се премахва от списъка.

Кликане върху елемент в списъка ще покаже информация за потенциалния проблем и ToDoItem таба в раздела с детайли и ще подчертае проблемната част от дизайна в червено.

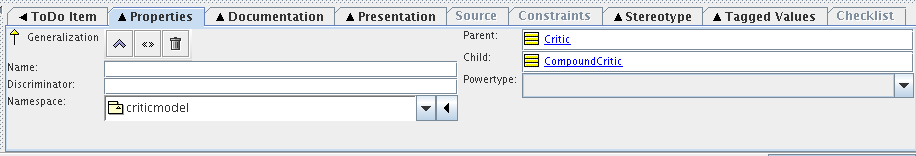
Когато се работи с голям дизайн, може да има голям брой проблеми, които трябва да бъдат разрешени преди дизайна да се завърши. Списъкът може да се групира по приоритет, тип и други.



**Фиг. 8.** *Раздел за предстоящи дейности*

### Раздел с детайли

Раздела с детайли (фиг. 9) позволява да се редактират детайлите на текущо селектирания елемент от дизайна или от списъка с предстоящи дейности. ToDoItem таба има стрелка на запад, за да индикира, че показва детайли за обект, селектиран в раздела за предстоящи дейности. Повечето от другите табове имат стрелка, сочеща на север, за да индикират, че показват детайли за обекти, селектирани в диаграмата или раздела за разглеждане.

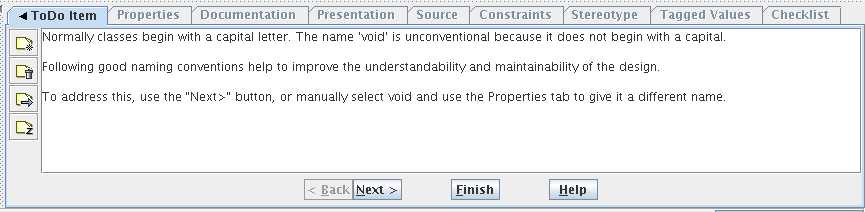
****

**Фиг. 9.** *Раздел с детайли*

#### Раздел ToDo Item

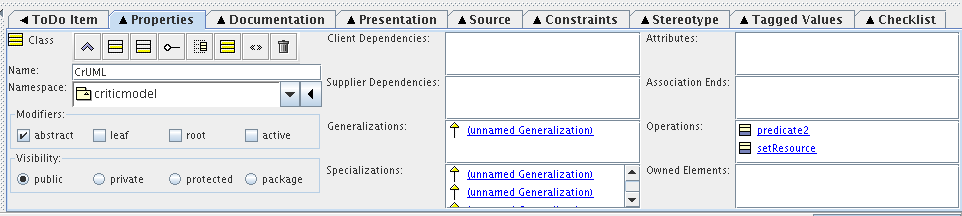
Раздела ToDoItem (фиг. 10) показва описанието на селектирания елемент от раздела с предстоящи дейности. Описанието се състои от три кратки параграфа за проблема, защо той е важен, и стъпки които могат да се предприемат за решаването му. Инструментите в панела, разположен отляво, позволяват да се:

* Добави нов елемент като лично напомняне
* Премахне текущия елемент
* Изпрати имейл на автора на критиката, генерирала текущия елемент
* Приспи критиката, генерирала текущия елемент: това изключва критката за 10 минути, и после автоматично я включва

**Фиг. 10.** *Раздел ToDO Item*

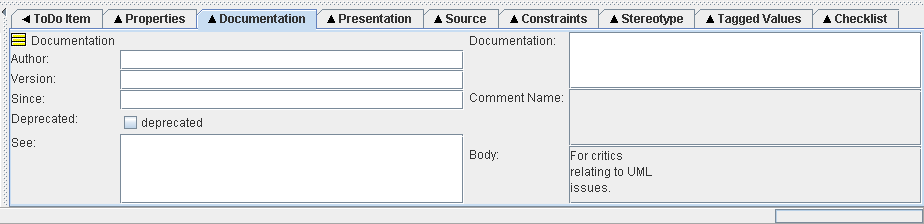
#### Раздел със свойства

Раздела със свойства (фиг. 11) показва свойствата на селектирания елемент от дизайна. Съдържанието на този раздел варира в зависимост от типа на селектирания елемент. Раздели със свойства са дефинирани за всички различни дизайн елементи. Освен това съдържат и различни бутони: например във фиг. 11 е изобразен раздел за клас, който има бутони за създаване на атрибути и операции.

**Фиг. 11.** *Раздел със свойства*

#### Раздел с документация

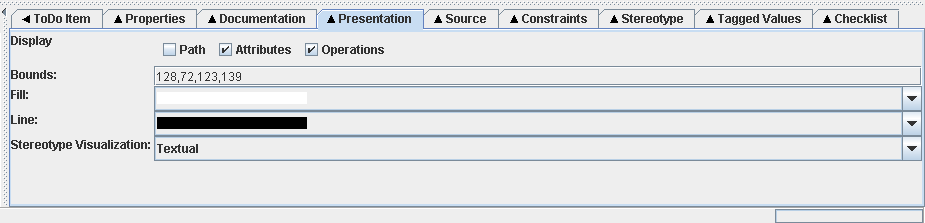
Раздела с документация (фиг. 12) позволява да се въведат различни типове документация за селектирания дизайн елемент. Документация може да бъде предоставена за всички типове дизайн елементи.



**Фиг. 12.***Раздел с документация*

#### Раздел за презентиране

Раздела за презентиране (фиг. 13) позволява да се редактират визуалните свойства на селектирания дизайн елемент. Съдържанието на този раздел може да варира значително в зависимост от селектирания елемент. Например, във фиг. 28, която е раздел за презентиране на клас, може да се промени видимостта на пътя пред името на класа, видимостта на разделите за атрибути и операции и други.

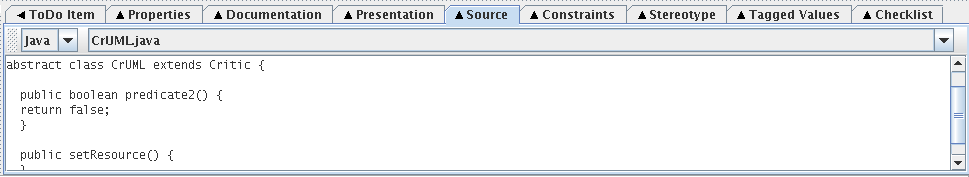


**Фиг. 13.** *Раздел за презентиране*

#### Раздел за сорс код

Раздела за сорс код (фиг. 14) позволява да се види кода, който ще се генерира за дадения елемент. Езика може да се избере от падащо меню. Второ падащо меню позволява избора на файл, там където се генерира повече от 1 файл(например С++).

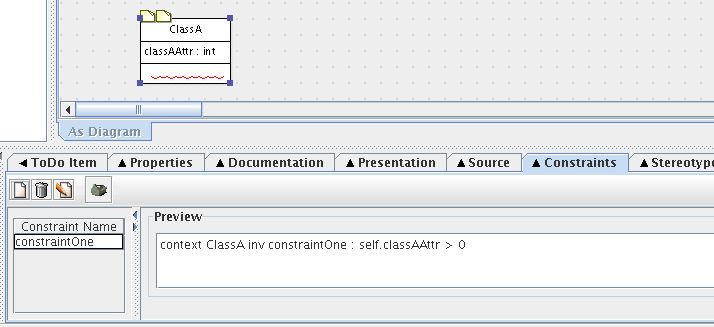
Стандартна инсталанция на ArgoUML позволява само Java. С помощта на плъгини, стават налични и езиците С++, PHP и C#.

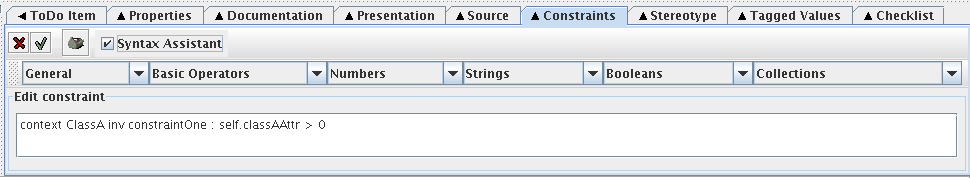


**Фиг. 14**. *Раздел за сорс код*

#### Раздел за ограничения

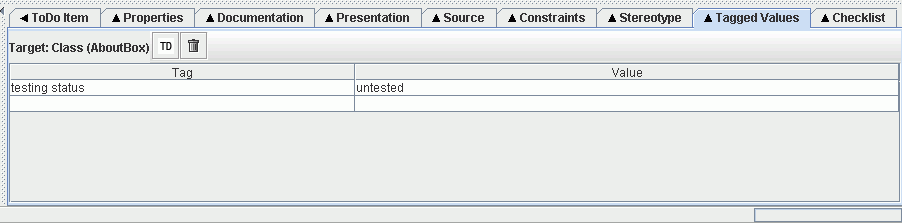
Раздела за ограничения (фиг. 15) позволява да се въведят и видят OCL ограниченията на текущия елемент. Object Constaint Language (OCL) е прост език, базиран на предикатната логика, който позволява да се добавя повече значение на диаграмата. Докато се редактира ограничението, Разделът за ограничения се превръща в редактор (фиг. 16).

**Фиг. 15.** *Раздел за ограничения*

**Фиг. 16.***Раздел за ограничения в режим редактор*

#### Раздел с тагове

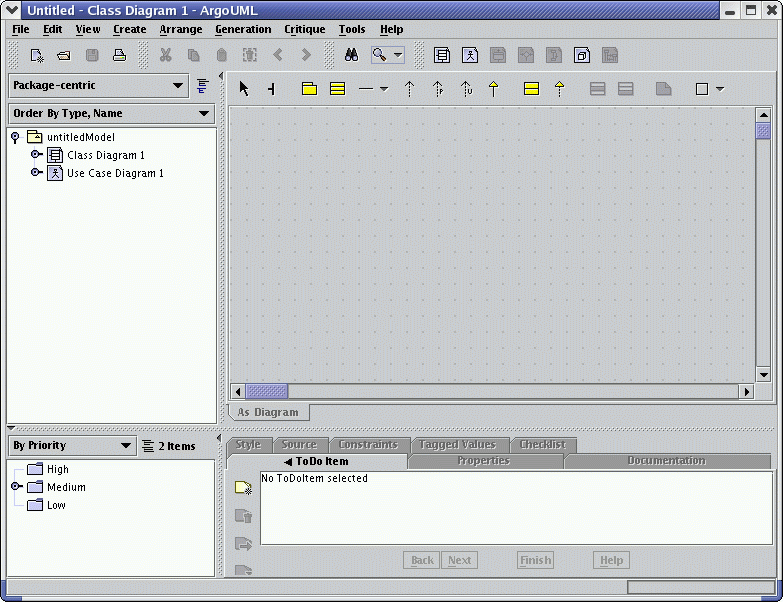
Раздела с тагове (фиг. 17) позволява да се въвеждат и видят таговете на селектирания дизайн елемент. Таговете са двойки ключ-стойност, които се пазят при дизайн елемента, но (обикновено) не се интерпретират от системата. Например ако искате да добавите индикация дали е тестван всеки клас от дизайна, може да се добави таг с ключ „testing status” и стойност измежду „untested”, “being tested” и „passed testing”.



**Фиг. 17.***Раздел с тагове*

## Създаване на Клас диаграма

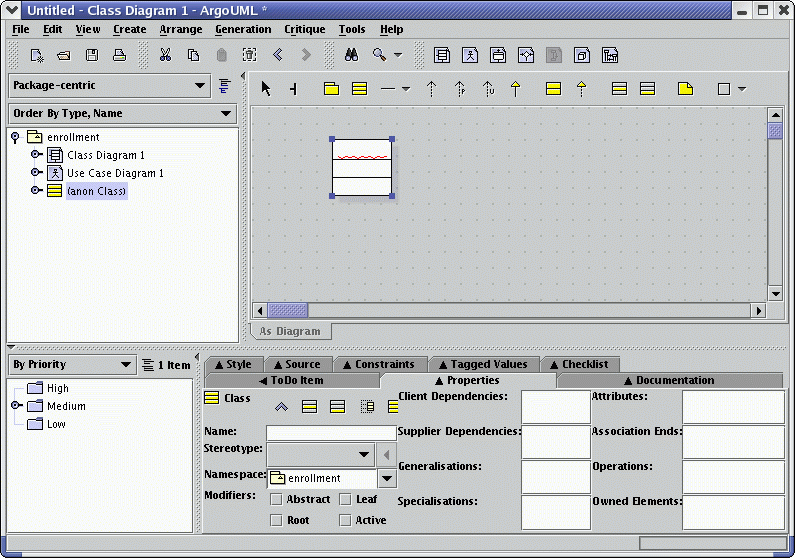
### Именуване на Проекта



**Фиг. 18.***Празен проект*

Имаме празен проект, от който да започнем ( фиг. 18). Сега е добро време да му дадем име (името по подразбиране е “Untitled”). За да именумава проекта, избираме File -> Save Project As…, и го запазваме под името Enroll.

### Именуване на Модела и Диаграмата

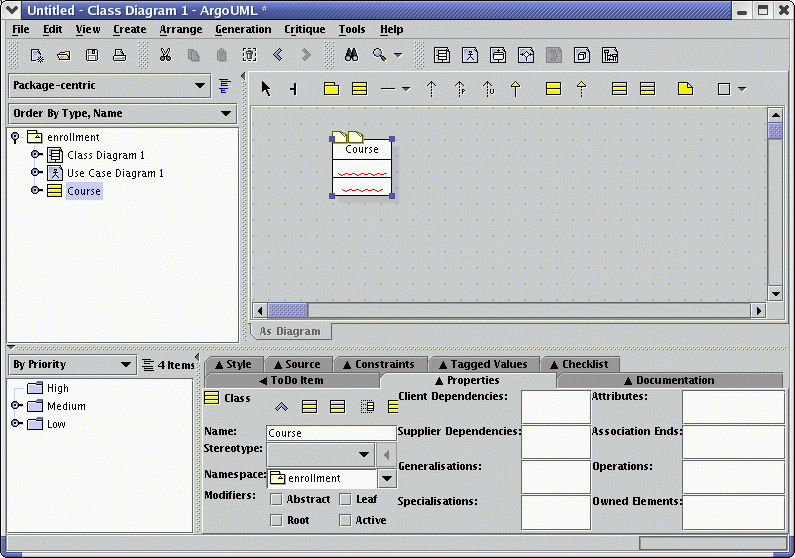


**Фиг. 19*.*** *Именуване на модела*

За да именуваме модела (пакета от най-високо ниво), избираме иконката на модела в раздела за разглеждане (вляво), и натискаме върху раздела със свойства (долу вдясно). Написваме „enrollment” в полето “name” (фиг. 19). По същия начин именуваме и клас диаграмата „main class diagram”.

### Създаване на Клас и Именуването му

За да добавим нов клас към диаграмата, натискаме върху клас иконката в лентата с инструменти http://argouml.tigris.org/tours/captures/classIcon.gif и след това натискаме някъде в редактора (фиг. 20). За да го именуваме, отиваме в раздела със свойства и въвеждаме името в “name” полето.

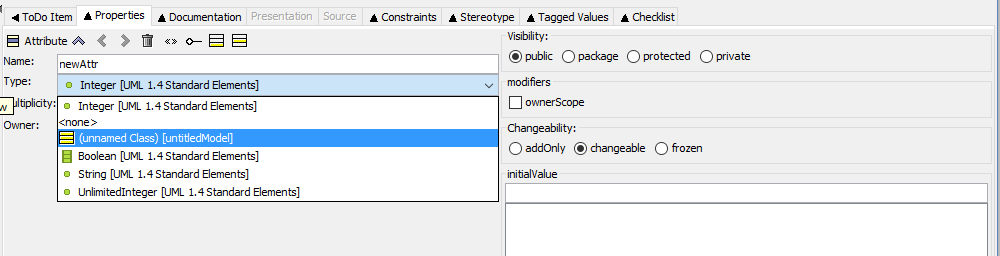


**Фиг. 20.***Създаване на клас*

### Добавяне на Атрибути

За добавяне на атрибути към класа, натискаме върху иконката “New Attribute” http://argouml.tigris.org/tours/captures/addAttributeIcon.gif в лентата с инструменти. За да преименуваме атрибута , можем да кликнем два пъти върху средния раздел на класа Course или да използваме полето Name в раздела със свойства.

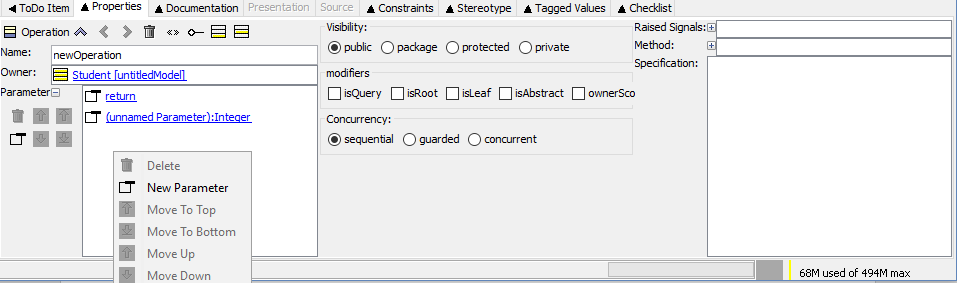
Можем да променим типа на атрибута чрез кликане два пъти върху него за да се покаже в раздела на свойстване. След това избираме типа от Type полето (фиг. 21).



**Фиг. 21***. Именуване и ибиране на тип на атрибут*

### Създаване на операция с атрибутите

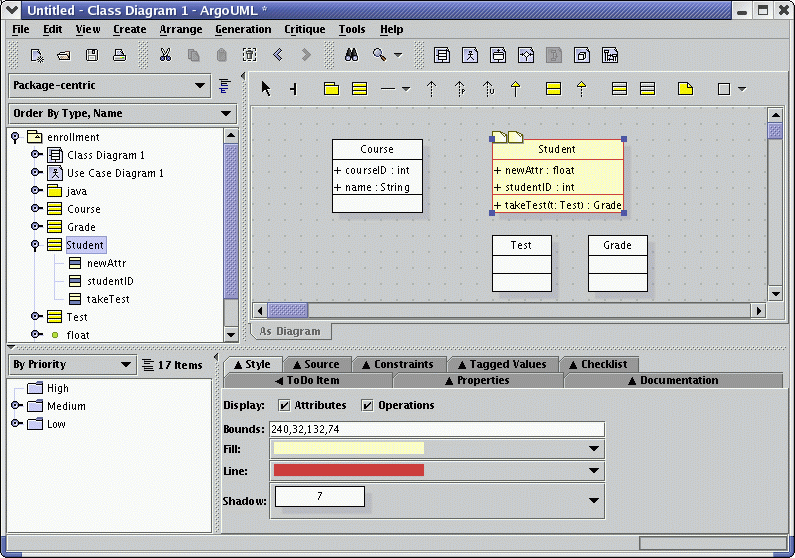
Нека добавим нов клас “Student” с атрибути “new Attr” и “studentId”. Добавяме нова операция чрез натискане на иконката на нова операцияhttp://argouml.tigris.org/tours/captures/addOperationIcon.gif в раздела със свойства. Параметрите на операцията се добавят със натискане на десния бутон в списъка с параметри в раздела на свойствата на операцията (фиг. 22). По същия начин задаваме и тип на връщане на операцията.



**Фиг. 22.** *Създаване на операция и добавяне на параметри*

### Използване на Стилове

Можем да сменим стила на Student класа. Нека направим цвета светло жълт и добавим червен контур. Това става през раздела за презентиране (фиг. 23).

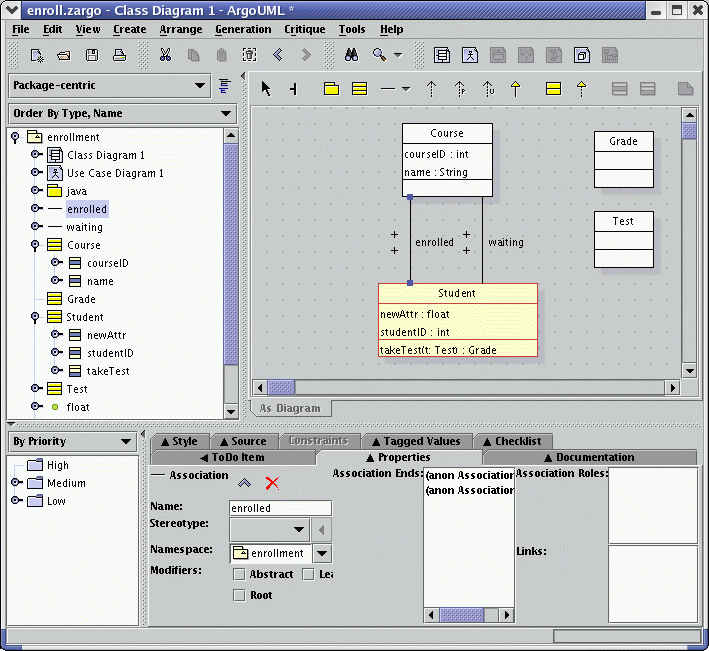


**Фиг. 23*.*** *Промяна на стила на клас чрез раздела за презентиране*

### Преместване, Преоразмеряване и Изобразяване на Асоциации

Преместване на клас се извършва чрез „влачене“ на класа във редактора. Преоразмеряване се извършва чрез селектиране на класа и влачене на синьото квадратче.

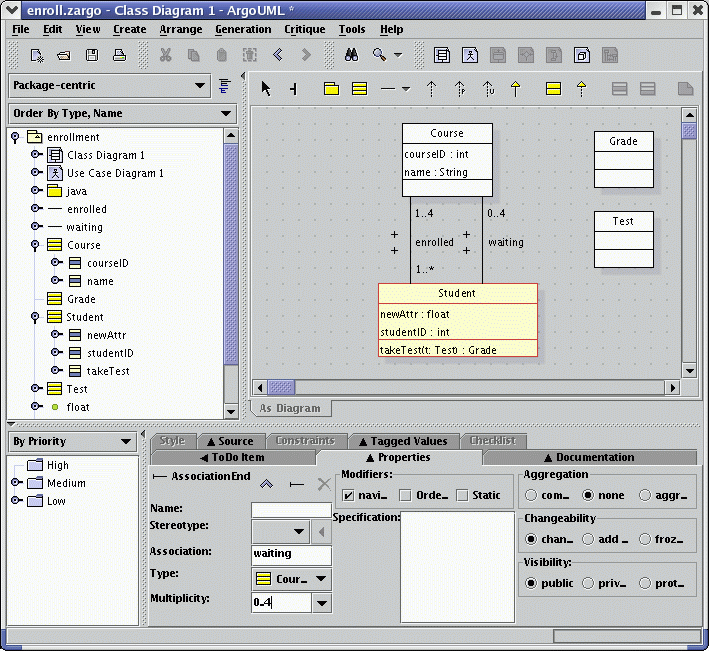
За добавяне на асоциация, натискаме иконката за асоциация в лентата с инструментитеhttp://argouml.tigris.org/tours/captures/associationIcon.gif. След това натискаме върху класа Course и влачим до класа Student. Асоциациите могат да бъдат местени и преоразмерявани по същия начин като класовете. Може да се добавят имена на асоциациите във раздела със свойства (фиг. 24).



**Фиг. 24.** *Създаване и именуване на асоциации*

### Добавяне на Множественост

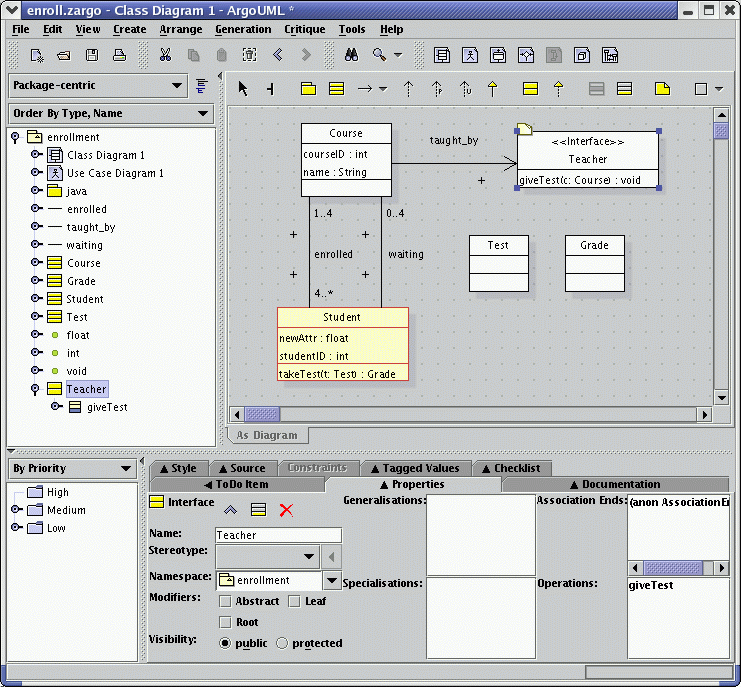
За да добавим множественост, селектираме асоциацията и натискаме два пъти върху съответния край. В раздела със свойствата въвеждаме стойността в полето “multiplicity” (фиг. 25).



**Фиг. 25.** *Добавяне на множественост*

### Добавяне на Интерфейс

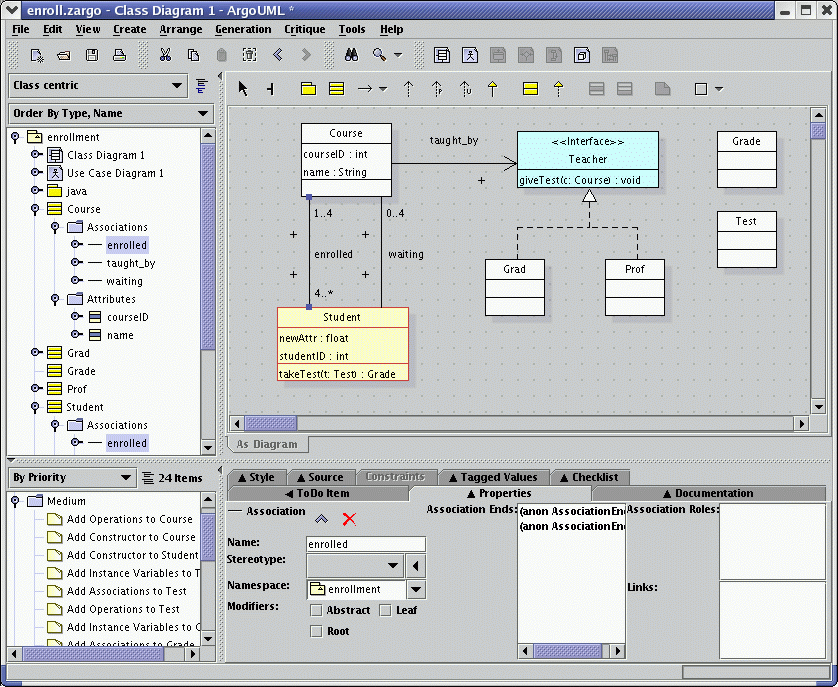
Избираме иконката Interface от лентата с инструменти и натискаме на свободно място в редактора. След това можем да дадем име по подобие на клас и да създадем асоциацииhttp://argouml.tigris.org/tours/captures/associationIcon.gif (фиг. 26). Забелязваме че асоциацията е в едната посока, спазвайки принципите на UML.



**Фиг. 26.***Добавяне на интерфейс*

### Добавяне на реализация на Интерфейс

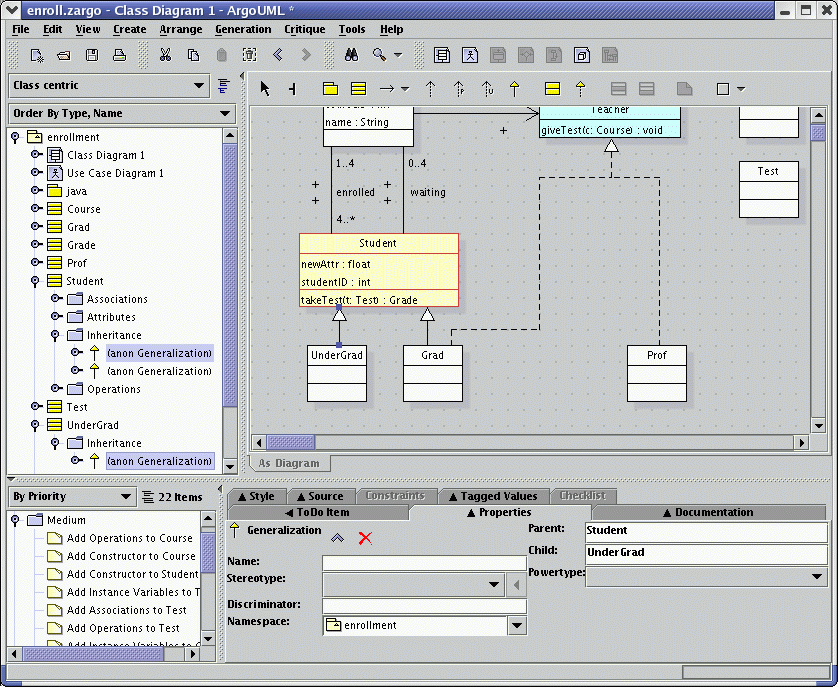
Нека оцветим интерфейса в светло синьо. Добавяме класовете Grad и Prof към диаграмата. После избираме Realise http://argouml.tigris.org/tours/captures/realizationIcon.gif от лентата със инструменти и го използваме по същия начин, по който създаваме асоциации (фиг. 27).



**Фиг. 27.** *Добавяне на реализация към интерфейс*

### Добавяне на генерализация

Ако искаме Grad и undergrad да наследяват Student, използваме Generalizationhttp://argouml.tigris.org/tours/captures/generalizationIcon.gif от лентата с инструментите (фиг. 28).



**Фиг. 28.***Добавяне на генерализация*

# Софтуерна архитектура на ArgoUML

Както е характерно за всички софтуерни системи, описаните функционални и нефункционални изисквания могат да бъдат реализирани по различни начини и макар да внасят ограничения, все пак позволят голям набор различни архитектурни решения. В тази секция ще разгледаме подхода, избран от самите разработчици на ArgoUML, като ще откроим онези моменти в архитектурата, които имат пряко отношение към спазването на нефункционалните изисквания, и ще направим оценка на тази архитектура – какви са силните и слабите й страни. В следващата глава ще предложим и наша, алтернативна архитектура, за която считаме, че би могла да се справи с някои от недостатъците на оригиналната.

В wiki-то на проекта ArgoUML специална страница [AUDSN] е посветена на бърз обзор на дизайн решенията, взети за продукта. Основна характеристика на архитектурата е избраният подход – целият софтуерен продукт е разделен на подсистеми, всяка от които има отговорности и подкомпоненти. Подсистемите са организирани в слоеве според зависимостите помежду им с цел ясно открояване на тези зависимости и избягване на потенциални проблеми, които биха могли да възникнат, ако допуснем циклични зависимости.

## Компоненти на архитектурата

Основните компоненти в ArgoUML са т.нар. подсистеми, които представляват относително самостоятелни модули, които реализират определена функционалност (т.е. удовлетворяват някое функционално изискване) или съществуват, за да позволят поддържането на качествен атрибут. Определихме ги като относително самостоятелни, защото все пак е възможно да има зависимости между тях, стига те да са съобразени с правилата на слоестата система, т.е. всяка система може да зависи само от системи в по-нисък от нея слой.

Характеризирайки компонентите на ArgoUML, за всеки от тях ще разгледаме следните му елементи:

* Име – всяка подсистема има име, което е свързано с основните й отговорности.
* Директория в структурата на проекта – всяка подсистема има единствена директория, в която се намират всички класове от нея, а отделните й подкомпоненти се намират в поддиректории. Това решение отразява идеята, че имплементацията на подсистема засяга само нея и смяна на версията, зависимостите или въобще вътрешната структура на подсистемата трябва да може да стане, без да бъдат засегнати останалите подсистеми
* Отговорности на подсистемата – всеки компонент се третира като отделна същност със своите отговорности, като тук ще се опитаме да откроим връзката между тях и изискванията към системата, които тя се опитва да спазва.
* Връзки с останалите подсистеми – ще се опитаме да откроим конекторите, свързващи отделните компоненти, както и това, по какъв начин подсистемата използва останалите, за да стане по-ясна цялостната картина на архитектурата на приложението
* API (Application Programming Interface) – това са всички публични и protected методи на всички публични и protected класове в съответната за подсистемата директория.
* Допълнителни елементи на подсистемата (ако има такива) – в текущата архитектура на ArgoUML някои от подсистемите има характерни особености – част от тях предоставят фасаден клас (малко по-подробно ще разкажем за това в следващата подсекция), а друга част имат интерфейс за плъгини.
* Имплементация на подсистемата – става въпрос за нейната структура, разделяне на подкомпоненти, интересни моменти в самата имплементация.

## Връзки между компонентите в ArgoUML

Подсистемите в ArgoUML могат да комуникират помежду си по няколко начина, от които най-често срещани са комуникацията през публично API, каквото имат всички подсистеми, и през фасадния клас за подсистемите, които имат такъв. Ако и двете възможности са налични, има възможност за избор дали да се използва публичното API или фасадния клас. В първия случай се създава зависимост между двете подсистеми, така че промяна във втората ще засяга и първата, докато във втория се създава зависимост само с фасадния клас, което носи известна гъвкавост – във втората подсистема могат да се внасят и по-сериозни проблеми, стига да не засягат интерфейса на въпросната фасада. Препоръчителна практика е да се дава насоки в документацията на дадена подсистема кога е удачно да се използва всеки от двата варианта.

Публичното API на подсистемите в ArgoUML е просто стандартен набор от публични методи, които могат да бъдат извиквани, както и публични променливи, когато това е разумно. То е задължително за всички подсистеми. Комуникацията чрез него се осъществява по стандартен начин – чрез извикване на методи с определени параметри.

Концепцията за фасаден клас е свързан с шаблона за дизайн Фасада, за който е характерно, че се създава клас, който обобщава предоставяната от дадена система функционалност и я прави публично достъпна за останалите подсистеми. По този начин се цели да се скрие, доколкото е възможно, голяма част от сложността на скритата зад фасадата подсистема и различните детайли, които може би са зависими от имплементацията й, без да имат пряка връзка с поведението, което тя предоставя, и отговорностите, които има. Макар не твърде широко разпространен в ArgoUML, този шаблон е характерен за продукта и фасадните класове в него предоставят най-вече най-често използваните функции на подсистемата, за да може да бъде използван само класът фасада. Той трябва да е по-стабилен и да се променя по-рядко от самата подсистема като цяло, както и да е добре документиран. Най-често представлява част от публичното API или негова опростена версия и е много добре документиран в самия код с помощта на Javadoc, за да са наясно разработчиците кога и за какво могат да го използват и кога не. Фасадните класове са разположени в директорията и пакета на съответната подсистема и комуникацията чрез тях отново се осъществява с помощта на извикване на методи с подаване на определени параметри.

Трети вариант за комуникация между подсистеми се осъществява с помощта на механизма на обратните извиквания (callbacks) – т.нар. събития (Events) или действия (Actions). В ArgoUML това е възможно с помощта на специални интерфейси, дефинирани от подсистемите, които поддържат този подход, в собствената им директория, съответно пакет. Събитията позволяват подсистемата да реагира на промени, настъпващи в други подсистеми или приложението като цяло, докато действията са механизъм, чрез който други подсистеми могат да заявят, че искат да бъдат извикани при настъпването на определени условия.

Не на последно място някои компоненти на ArgoUML предоставят и още едно средство за конекция – интерфейси за плъгини. Това са най-често фасадни класове, където други модули или плъгини могат да се свържат и да променят или допълнят поведението на подсистемата. Това придава гъвкавост на архитектурата, тъй като лесно може да се добавят или премахват функционалности – в ArgoUML това става дори без нужда от преинсталация в общия случай, просто с редактиране на конфигурационните файлове и рестартиране на приложението.

Както споменахме, подсистемите в ArgoUML са разделени на слоеве, за да са по-ясни механизмите на комуникация между тях. Следва да разгледаме по-подробно тези слоеве.

## Общ преглед на архитектурата



**Фиг. 29.***Общ изглед на архитектурата – диаграма тип C&C*

Както споменахме, подсистемите в ArgoUML са разделени на слоеве, за да са по-ясни механизмите на комуникация и зависимостите между тях. Това става видно и от представанета диаграма на компонентите и конекторите, която предоставя птичи поглед върху системата. Следва да разгледаме в детайл отделните й елементи, характеризирайки ги по описания по-горе модел. За целта ще разгледаме слоевете така, както са дефинирани в официалната документация на дизайна на продукта: слой на подсистемите от ниско ниво, слой на подсистеми за преглед и контрол, слой на подсистемите от високо ниво, слой на подсистемите, които могат да бъдат зареждани динамично, и външни библиотеки, използвани от продукта.

Тук е моментът да отбележим, че тази организация всъщност следва добре познатият архитектурен модел MVC – Model View Controller. При него имаме модел, който представя света на приложението и съдържа основните обекти, с които борави приложението, изглед или view, който представлява интерфейса, предоставен на потребителя, и контролер, който, въз основа на входа от view-то, извършва промени в модела, като се съобразява с бизнес логиката. В случая на ArgoUML моделът е обособен в подсистема, която е в слоя за ниско ниво, тъй като не зависи от почти никоя друга подсистема, а изгледът и контролерът се намират в слоя за изглед и контрол.

## Слой на системите от ниско ниво

Това са основни, инфраструктурни подсистеми на ArgoUML, които са налични и могат да бъдат използвани от всички останали подсистеми на продукта. Те имат проста структура на зависимостите: всяка друга може да зависи от тях, но няма зависимост между тях и подсистема от по-висок слой, както и между отделните подсистеми. Именно това ги отличава – не зависят от никой друг пакет. Тук спадат няколко подсистеми, които можем да видим изобразени на диаграмата на фиг. 30.



**Фиг. 30.***Изглед към подсистемите от ниско ниво*

### Подсистема за интернационализация (internationalization)

**Местоположение**: Подсистемата за интернационализация се намира в пакета org.argouml.i18n

**Отговорности**: Тази подсистема предоставя функционалност за превеждане на символни низове от един естествен език на друг. Допълнително тя позволява добавянето на нови езици, менажира избор на език, включително дефинира език по подразбиране.

**Връзка с други компоненти и особености:** Като подсистема от ниско ниво, тя не зависи от никоя от останалите подсистеми. За сметка на това всички други подсистеми могат да я използват. Основният начин, по който се осъществя това, е чрез директно извикване на методи. Характерно за подсистемата обаче е това, че тя предоставя фасаден клас – org.argouml.i18n.Translator. На практика всички останали подсистеми комуникират с нея чрез него.

**API:** Основната функционалност, която предоставя тази подсистема, е представена именно чрез фасадния клас. Това са общо взето три основни групи функционалности: за инициализация (зареждане на подсистемата, зареждане на файлове с преводи и т.н), избор на локал (достъпване на списъци с наличните локали, запитване за локали по подразбиране, експлицитен избор на локал) и локализация на символни низове (включително е възможна локализация на по-сложни шаблони, със форматиране и допълнителни аргументи на превежданото съобщение).

**Имплементация**: Механизмът на работа на тази подсистема е проектиран с цел простота – вместо символни низове потребителите трябва да използват ключове, а локализаторът индексира с тези ключове в асоциативен масив, избран въз основа на текущия език. Що се отнася до езика по подразбиране, той се определя въз основа на подаден команден аргумент или потребителска конфигурация, а при отсъствието на такива – по настройките на локализация на потребителския компютър. Ако по някаква причина и те отсъстват или са недостъпни, езикът по подразбиране е американски английски език.

### Подсистема за менажиране на задачи

**Местоположение**: Подсистемата за менажиране на задачи се намира в пакета org.argouml.taskmgmt

**Отговорности**: Тази подсистема подпомага менажирането на работата на продукта, когато са стартирани множество независими задачи. В момента това означава главно следене на прогреса – ако е стартиран дълъг процес в отделна нишка, тази подсистема следи прогреса му, за да го визуализира в потребителския интерфейс.

**Връзка с други компоненти и особености:** Като подсистема от ниско ниво, тя не зависи от никоя от останалите подсистеми. За сметка на това тя е в директна комуникация с две други подсистеми – тези за потребителския интерфейс и за модела. Комуникацията се извършва основно с помощта на събития – подсистемата за менажиране на задачи приема събития от модела през магистралата за събития, както и от други канали, и генерира събития, с които уведомява подсистемата за потребителския интерфейс, за да се обновят съответните елементи от UI-a.

**API:** Основната функционалност, която предоставя тази подсистема, е представена чрез класа ProgressMonitor. Той съдържа функционалности за избор на задачи, нотификация за прогрес и нотификация за съобщения. При създаването на по-сложна задача е достатъчно да се пази референция към този обект и да се викат неговите методи.

**Имплементация**: Механизмът на работа на тази подсистема е прост – нотификацията се осъществява с помощта на събития от типа ProgressEvent, които носят текущ прогрес и общ размер на задачата. Те се превеждат от подсистемата до подходящ формат за системата на графичния интерфейс.

### Подсистема за конфигурация

**Местоположение**: Подсистемата за конфигурация се намира в пакета org.argouml.configuration

**Отговорности**: Тази подсистема се грижи да съхранява и прочита всички конфигуриуеми аспекти на приложението, съхранявани в специален файл с настройки. Тя обаче не се грижи за конфигурационните настройки на самите проекти – те са отговорности на класовете Project и ProjectSettings, които са част от подсистемата на приложението.

**Връзка с други компоненти и особености:** Като подсистема от ниско ниво, тя не зависи от никоя от останалите подсистеми. За сметка на това тя е в директна комуникация с останалите подсистеми, като това може да стане по два основни механизма. Първият е чрез директно извикване на методи – подсистемите, които се интересуват от някое свойство, могат директно да запитат за стойността му. Вторият е с помощта на събития – всяка подсистема може да се регистрира за информация за промени в определени свойства. Освен това тази подсистема достъпва и конфигурационните файлове, които се намират във файловата система.

**API:** Основната функционалност, която предоставя тази подсистема, е представена чрез класа Configuration. Той съдържа функционалности за зареждане на конфигурация от файлове, записване на конфигурация във файлове, достъпване на стойността на даден ключ, промяната й, добавяне и премахване на listeners – наблюдатели за събития.

**Имплементация**: Механизмът на работа на тази подсистема е подобен на този на системата за интернационализация – всяко свойство се идентифицира с ключ, стойностите се четат и записват в асоциативен масив, който се сериализира и десериализира от диска. Допълнителната особеност тук е, че ключовете са йерархични – състоят се от няколко идентификатора, разделени с точка. Това подпомага по-ясната семантика, добро логично обособяване на задачите и директно отговаря на структурата на конфигурационните файлове, в които са записани съответните свойства.

### Подсистема за персистентност (persistence)

**Местоположение**: Подсистемата за персистентност се намира в пакета org.argouml.persistence

**Отговорности**: Тази подсистема се грижи да съхранява и прочита всички данни от другите подсистеми на програмата от някаква среда за съхранение. Това включва всички файлове с потребителски данни, както и parse-ването на използваните формати, когато това е необходимо.

**Връзка с други компоненти и особености:** Като подсистема от ниско ниво, тя не зависи от никоя от останалите подсистеми. За сметка на това тя е в директна комуникация с останалите подсистеми, като това става главно с извикване на методи. Освен това тази подсистема извършва най-интензивна комуникация с файловата система, четейки и записвайки множество файлове в редица формати, някои от които компресирани.

**API:** Основната функционалност, която предоставя тази подсистема, е предоставена за другите системи през класа PersistenceManager. Той съдържа функционалности за зареждане на специални класове, наречени persisters – всеки от тях предоставя функционалности за съхраняване и зареждане на данни.

**Имплементация**: Механизмът на работа на тази подсистема е базиран на принципите на абстракцията – единен интерфейс за записване и четене от файл е имплементиран от множество класове, наречени persisters. Някои от тях се грижат за сериализацията на сложни обекти – диаграми, фигури, модели, други се грижат за работа със специфични формати – XML например, а трети се грижат за parse-ването на формати – например ArgoParser за формата .argo, zargo парсър за формата .zargo, който е компресиран, и така нататък. Тази структура на системата позволява максимална гъвкавост – използваният формат за сериализация и десериализация е неизвестен за останалите подсистеми и много лесно се добавят нови такива.

### Подсистема за logging

**Местоположение**: Подсистемата за logging нe e нов пакет в ArgoUML, а е външна библиотека – използва се директно log4j, което е може би най-популярната Java библиотека за тази цел.

**Отговорности**: Тази подсистема се грижи да предоставя функционалността за записване на съобщения, които носят информация за важни моменти в работата на програмата. Те се използват както в ситуации, когато има проблем, грешка или бъг и трябва да бъде установена причината, така и при нормално опериране на програмата, ако искаме да проследим хода й или търсим за появата на определени събития.

**Връзка с други компоненти и особености:** Като подсистема от ниско ниво, тя не зависи от никоя от останалите подсистеми. Всички те обаче я използват, като конекцията се осъществява с помощта на директно извикване на методи. Освен това тази подсистема е в релация и с файловата система, тъй като съответните съобщения се записват в log файлове на диска.

**API:** Основната функционалност, която предоставя тази подсистема, се достъпва през класа LOG. Той има по един метод за всяко ниво на съобщение, поддържано от системата – debug, info, warn, error, fatal, подредени в този ред по приоритет от най-нисък към най-висок. FATAL са съобщения, асоциирани с много сериозни грешки, резултатът от които е прекратяване на работата на приложението, и обикновено съдържат много подробна информация за грешката. ERROR също са съобщения за грешка и носят много подробна информация за грешката, но за разлика от FATAL е възможно приложението да се възстанови от нея. WARN е предупреждение за потенциално проблемна ситуация, но реално не е възникнала грешка, когато го използваме. INFO носи информация за хода на работа на приложението, но на високо ниво – само съществени събития се обозначават тук, като зареждане на модули и подсистеми, зареждане и запаметяване на файлове, важни модификации. DEBUG носи много по-подробна информация за множество събития, които не са толкова важни сами по себе си, но са полезни при дебъгване на приложението

**Имплементация**: Работата на тази подсистема разчита изцяло на външната библиотека log4j. Където е необходимо, се създава статичен за класа logger, който сочи към определен файл, и се извикват неговите методи, за да се запишат съобщения със съответното ниво. Важно е да се отбележи, че logging подсистемата е изключена по подразбиране, но може да се активира с помощта на команден параметър при извикването на ArgoUML – става въпрос за опцията log4j.configuration, с която може да подадем път към .lcf файл, съхраняващ цялата конфигурация на logging системата.

### Подсистема за модела

**Местоположение**: Намира се в пакета org.argouml.model.

**Отговорности**: Тази подсистема съдържа в себе си цялото знание за модела, който се използва (в общия случай това е UML) и предоставя консистентен интерфейс за манипулирането му, като същевременно скрива от другите подсистеми на ArgoUML каквото и да е знание относно това с цел да се постигне по-добро разделяне на отговорностите и абстрахиране от конкретния използван модел.

**Връзка с други компоненти и особености:** Като подсистема от ниско ниво, тя не зависи от никоя от останалите подсистеми. Голяма част от останалите системи обаче я използват, като конекцията се осъществява по няколко начина. Първият от тях е чрез извикване публичните методи от API-то й – всяка подсистема може да извика директно помощни функции или функции за създаване на отделни елементи от модела. Много често обаче комуникацията се осъществява отново чрез извикване на метод, но този път през фасадата на подсистемата – тук имаме фасаден клас Model, фасадни фабрики и Helper класове, които обобщават функционалностите и често са по-удобни и разбираеми за употреба. Съществува и трети канал за комуникация – всяка промяна в модела поражда събитие, което се разпространява по event bus-a. Всяка подсистема от по-високо ниво може да се регистрира да получава нотификации за различни събития, за да може промяна в модела да се отрази едновременно във всички изгледи, перспективи и допълнителни подсистеми, които изграждат сложната екосистема на ArgoUML. Често срещана практика е отделните подсистеми да обновяват модела чрез директно извикване на функции или през фасадата, а същевременно да служат по магистралата за обновления, предизвикани от други компоненти. Така се осъществява връзките с компонентите за диаграмите, панелите със свойства и редица от плъгините като тези за генериране на модели по код (reverse engineering), за прилагане на ограничения и т.н. Дори някои компоненти, които не променят директно модела, също слушат по магистралата за обновления – такива са компонентите за нотация и за Explorer-a, дизайн критиците и други. Играейки именно ролята на модела в MVC шаблона за дизайн, този компонент е един от най-важните и е централен за ArgoUML.

**API:** Основната функционалност, която предоставя тази подсистема, може да се достъпи през фасадните класове, които добре обобщават функционалността й. Основен клас в този смисъл е класът Model, който предоставя достъп през интерфейса си до останалите елементи на тази подсистема. Главната допълнителна функция, която той изпълнява в допълнение към абстрахирането на модела, е налагането на правилата за добро оформление, дефинирани в UML стандарта – така се гарантира в някакъв смисъл коректност и консистентност на модела. Останалата функционалност се предоставя от класовете фабрики, които предоставят всички методи за създаване на елементи на модела, и помощните класове, които предоставят методите за манипулация – например прочитане и промяна на свойства. Магистралата за събития е представена чрез ModelEventPump интерфейс, който предоставя функционалност за регистрация за промени в модела – така наречените PropertyChangeEvent, които заменят конкретните събития, генерирани от модела.

**Имплементация**: В най-новата версия на ArgoUML, която разглеждаме, имплементацията използва Netbeans MDR (Metadata Repository) библиотеката, но съществуват и имплементации, базирани на NSUML библиотеката, на Eclipse UML2 инструментите и други. Благодарение на добрата абстракция, заложена в архитектурата, те могат да бъдат заменяни прозрачно за останалите компоненти и съответно има методи за избора им в API-то на фасадния клас Model. Сега ще се спрем на класовете фабрики, помощните класове и event магистралата.

Класовете фабрики са стандартни за въпросния шаблон за дизайн – те съдържат create методи за всички елементи на модела. Обикновено съществуват различни build методи в допълнение, които имплементират често срещани ситуации на създаване на елементи, като това са съобразени с правилата за оформление на UML. Използването на фабричните методи е важно, защото освен правилното конструиране на обекта гарантира и правилна настройка на системата за обработка на събития.

Помощните класове съдържат методи за манипулация на елементите на модела – например за вземане на всички елементи от даден тип от модела, за достъп и промяна на свойства. За всеки елемент от UML спецификацията съществуват съответен абстрактен клас с дефиниция, Helper клас с помощни методи и Factory клас с функции за построяване. Що се отнася до самите елементи, всеки от тях си има вътрешно ID, наречено MOF ID, менажирано от модела, и се сериализира в XMI файлове (за това се грижи системата за персистентност).

Магистралата за събития е основното средство за комуникация между елементите на модела и остатъка от ArgoUML. Чрез нея събитията от модела се прихващат и препращат към останалите подсистеми. Регистрацията на заинтересовани от събитията подсистеми се случва тук – може да се регистрираме само за някои или за всички събития.

### Подсистема за UML профили

**Местоположение**: Подсистемата за UML профили се намира в пакета org.argouml.profile

**Отговорности**: Тази подсистема се грижи да предоставя поддръжка за употребата, дефиницията и управлението на UML профили в ArgoUML. Те представлят дефинирани от стандарта средства за разширяване на ArgoUML чрез дефиниране на т.нар. стереотипи (нови типове елементи, разширяващи някой от оригиналните, но внясащи допълнителна семантика), тагове и ограничения. В ArgoUML профилите могат да имат и асоциирани критици, които да проверяват и помагат за спазването на внесените допълнителни ограничения.

**Връзка с други компоненти и особености:** Системата за UML профили се използва от подсистемите от високо ниво. В много отношения тя прилича на подсистемата за моделите и наистина между двете има и семантична връзка – в някакъв смисъл системата с профилите е разширение на тази за модела. Съответно по подобен начин системите от високо ниво пряко извикват методи на системата за профилите и могат да се регистрират за събития. Най-често се използва фасадният клас, който предоставя достъп до класовете Profile и ProfileManager.

**API:** Основната функционалност, която предоставя тази подсистема, се достъпва през фасадния клас ProfileFacade. Той предоставя от своя страна достъп до класовете ProfileManager и отделните инстанции на Profile. Те позволяват динамично регистриране и премахване на регистрани профили, както и всякаква друга функционалност, свързана с менажиране на въпросните профили.

**Имплементация**: Основната концепция, на която се основава тази подсистема, е именно концепцията за профил. Профилите обикновено се дефинират като XMI файл, в който се добавят дефиниции на стереотипи и тагове, които специализират мета-класовете от UML модела. Често профилите имат и асоциирани критици, които валидират потребителската диаграма въз основа на правилата, дефинирани в профила. Профилът може да съдържа и Formatting strategy, която предоставя специално форматиране, FigNodeStrategy, която предоставя фигури, които да заместят текстовото описание на стереотипите на профила, и DefaultTypeStrategy, която дефинира типове по подразбиране за атрибутите, параметри и стойностите на връщане за профила. Критиците пък могат да се дефинират или чрез наследяване на клас, или с помощта на правила, описани в OCL (Open Constraint Language), които се прилагат от предефинирания критик CrOCL. Така профилите реално често носят със себе си и потребителски дефинирани ограничения. Освен това те мотаг да имат зависимости към други профили.

Самата подсистема за работа с профили не само позволява реализирането на тази функционалност, но и предоставя средства за менажиране на профили. Това включва на първо място регистрирането им – то може да стане по подразбиране, тъй като някои профили се зареждат от мениджъра още при създаването му, чрез зареждане на дефиниции от XMI файлове, специфично посочени от потребителя, или чрез плъгини – всеки плъгин може да дефинира списък от профили и критици, които ще бъдат заредени заедно с него. Профили може да добавим и ръчно, като наследим Profile класа и ги заредим заедно с другите вградени профили при инициализацията на тази подсистема. Профили могат да се добавят и премахват на ниво проект.

## Слой за преглед и контрол

Всички подсистеми в него разчитат на подсистемите от по-долния слой, най-вече на подсистемата, представяща модела, и тясно свързаната с нея подсистема за менажиране на профили. Тук са функционалностите, грижещи се за контрол на работата и визуализация, т.е. View и Control елементите от шаблона MVC. Взаимовръзките между тях и останалите подсистеми може да видим на фиг. 31.



**Фиг. 31.***Изглед към подсистемите за преглед и контрол на диаграмите*

### Подсистема за диаграми

**Местоположение**: Подсистемата за UML диаграми се намира в пакета org.argouml.diagram

**Отговорности**: Тази подсистема има задачата да представя диаграмите на потребителя и да му предостави възможност да ги манипулира през различни изгледи. Тя трябва да гарантира, че направените от него промени веднага ще бъдат отразени в модела и ще се разпространят и до останалите подсистеми с помощта на event bus-a.

**Връзка с други компоненти и особености:** Подсистемата за диаграмите е система от междинния слой и зависи от тези от по-ниския слой – те й предоставят функционалности за интернационализация, персистентност, конфигурационни опции, logging и така нататък. От системите от ниско ниво най-голяма е зависимостта й от подсистемата за модела (съответно и тази за профилите). Подсистемата за диаграмите е свързана с тях по два начина. Първо, чрез директни извиквания на методи тя отразява промените, направени от потребителя, в модела – създавайки и премахвайки елементи от диаграмата, промяна на някои свойства като местоположение, създаване на нови диаграми и така нататък. Второ, чрез системата за събития, регистрирайки се за релевантните събития – промени в модела, за да може винаги представената диаграма като изглед да е актуална. Освен това има и пряка връзка с подсистемата за нотация – чрез извиквания на методи от нея се взима текстовото описание, което съпътства диаграмата като обяснение на изобразените в нея елементи. Комуникация чрез действия се осъществява с подсистемата за потребителски интерфейс – при определени условия и потребителски вход системата за GUI извиква регистриралата се подсистема за диаграми, за да ги обработи.

**API и имплементация:** Тази подсистема се характеризира имплементационно с използването на GEF – Graph Editing Framework, за представяне на самата диаграма. Основните ключови класове за тези за модела на графа, фигурата, която представя елемент от графа с определена визуализация, и селекцията – тя маркира кои елементи са избрани в даден момент, да за им се приложат групови действия, когато това е възможно.

Класът Fig e особено важен, защото благодарение на него се представят съществуващите и могат да се добавят нови елементи в диаграмат, стига да се добавят към графовия модел и т.нар. Renderer. Когато добавим и съответен панел със свойства, получаваме завършен елемент на диаграмата. За да може да фигурира в даден тип диаграма, достатъчно е да фигурира в съответния й GraphModel клас. За да се визуализира коректно, се използват DiagramRenderer клас и наследниците му, които предоставят методи, с помощта на които по даден UML обект може да получим фигура, която го визуализира. Тази фигура може да се обвърже с обект, който да й предоставя нотация, и с определени действия в потребителския интерфейс, за да може да бъде създаван и модифициран директно от потребителя

### Подсистема за панели със свойства

**Местоположение**: Подсистемата за панели със свойства се намира в пакета org.argouml.umlpropertypanels.

**Отговорности**: Тази подсистема има задачата да покаже алтернативен изглед към диаграмите и обектите в модела, както и да представи модифицируемите свойства на тези обекти, за да може потребителят да ги променя. Тези промени трябва да се отразят веднага върху съдържанието на модела.

**Връзка с други компоненти и особености:** Подсистемата за панелите със свойства е система от междинния слой и зависи от тези от по-ниския слой – те й предоставят функционалности за интернационализация, персистентност, конфигурационни опции, logging и така нататък. От системите от ниско ниво най-голяма е зависимостта й от подсистемата за модела (съответно и тази за профилите). Подсистемата за панелите със свойства е свързана с тях по два начина. Първо, чрез директни извиквания на методи тя отразява промените, направени от потребителя, в модела – тук става въпрос най-вече за промяна на различни свойства, като това, какви свойства може да променим, зависи от самия тип на обекта. Второ, чрез системата за събития, регистрирайки се за релевантните събития – промени в модела, за да може винаги представените допълнителни изгледи да са актуални. Комуникация чрез действия пък се осъществява с подсистемата за потребителски интерфейс – при определени условия и потребителски вход системата за GUI извиква регистриралата се подсистема за панели със свойства, за да ги обработи.

**API и имплементация:** Ключов за тази подсистема е класът PropPanel, чийто наследници представят съответните панели за свойства на различните типове UML елементи. Благодарение на GUI системата във всеки един момент е известно кой или кои са селектираните елементи и подсистемата за панелите със свойства винаги се обновява, така че да предостави информация за конкретно избрания или избраните в момента елементи. Самият панел пък изглежда като таблица с полета и разделители между тях, като са предоставени методи за добавянето и премахване на полета и сепаратори, а също и на бутони с допълнителна функционалност (към тях дори може да се прикачват действия). Самите полета могат да са падащ списък, редактируем текст, радио бутони или чек списъци, списъци, комбинирани полета, текстови области и въобще голяма част от характерните за съвременен графичен потребителски интерфейс контроли.

### Подсистема за преглеждане на елементи (explorer)

**Местоположение**: Подсистемата за преглеждане на елементи се намира в пакета org.argouml.ui.explorer

**Отговорности**: Тази подсистема има за цел да предоставя изгледи към модела, върху който работи потребителя, заедно с всички елементи в него – диаграми, други обекти, като става въпрос най-вече за изгледи във вид на дърво, които ясно отразяват йерархията на елементите в системата. Важна задача за него е да поддържа както разнообразни изгледи, които да предоставят полезни за потребителя нови перспективи върху системата, така и да поддържа консистентността между тези подсистеми, като реагира и на директен потребителски вход, и на промени в системата поради други причини (например промяна в друг изглед).

**Връзка с други компоненти и особености:** Подсистемата за преглеждане на елементи е система от междинния слой и зависи от тези от по-ниския слой – те й предоставят функционалности за интернационализация, персистентност, конфигурационни опции, logging и така нататък. От системите от ниско ниво зависи от подсистемата за модела (съответно и тази за профилите). В общия случай тя не може да променя директно модела, тъй като има за цел просто да предоставя изгледи, но разчита на event bus-a, от който получава нотификации за промени в модела, за да обнови изгледите, които е представила пред протребителя в момента. Комуникация чрез действия пък се осъществява с подсистемата за потребителски интерфейс – при определени условия и потребителски вход системата за GUI извиква регистриралата се подсистема за преглеждане на елементи, за да ги обработи. Тук комуникацията е особено интензивна, тъй като потребителят често взаимодейства през потребителския интерфейс с тази подсистема – разпъва и сгъва елементи в дървото, променя перспективи, подрежда ги според различни наредби, селектира различни елементи и така нататък.

**API и имплементация:** Тази подсистема предоставя разнообразни функционалности – добавяне и премахване на перспективи, избор на перспектива, които се предоставят от класа PerspectiveManager, както и работа с правила за навигация, известни като PerspectiveRules – отделни класове, които дефинират стандартни форми на навигация като например отиди при родителя, проследи преход до друг родител, смени диаграмата и така нататък. Освен това са налице класове от тип Ordering, които дефинират подредба на елементите в перспектива, като може да се дефинират и нови такива, наследявайки базовия клас. За самата визуализация се използват ExplorerTree и ExplorerTreeNode, които използват мързелив подход при изграждането й, а за коректното отразяване на нотификации, идващи от модела, се грижи класът ExplorerTreeModel. Не на последно място трябва да отбележим, че PerspectiveRules са особено мощна концепция – те позволяват дори да изградим нови перспективи от нулата, като просто дефинираме правило, по което от възел да се генерират наследниците му.

### Подсистема за нотация

**Местоположение**: Подсистемата за UML диаграми се намира в пакета org.argouml.notation

**Отговорности**: Тази подсистема има задачата да поддържа генериране, редактиране и обновяване на текстово представяне на един или повече елементи на UML модела, като това може да стане на различни езици. Такива елементи с нотация обикновено се прикачват към диаграмите и елементите на системата за преглед.

**Връзка с други компоненти и особености:** Подсистемата за нотация е система от междинния слой и зависи от тези от по-ниския слой – те й предоставят функционалности за интернационализация, персистентност, конфигурационни опции, logging и така нататък. Функционалността за интернационализация е от особено значение тук. От системите от ниско ниво най-голяма е зависимостта й от подсистемата за модела (съответно и тази за профилите). Подсистемата за нотация се регистрира в event bus-a, за да получава обновления от тях. Така при промяна в модела динамично се променя и текстовото описание, както и очаква да се случи потребителят. Връзка съществува и със системите за преглед и най-вече тази за диаграмите – с помощта на директно извикване на методи те получават текстовото си съдържание от асоциирания NotationProvider обект с помощта на метода toString().

**API и имплементация:** Основната особеност на тази система е разделението на отговорностите между две основни групи класове. Първата група са класовете за нотация, които имат toString() метод и определени свойства и са асоциирани с определен език, който се определя на ниво проект и най-често дори се записва в настройките на проекта. За предоставянето на тези нотации се грижат класовете, наследници на NotationProvider, които имплементират най-различни възможни комбинации от езици и текстови представяния. Този модел е доста гъвкав – например възможно е допълнителни източници на нотация и дори езици за нотацията да се дефинират чрез плъгини.

### Подсистема за графичен потребителски интерфейс (GUI)

**Местоположение**: Подсистемата за UML диаграми се намира в пакета org.argouml.ui

**Отговорности**: Тази подсистема има задачата да предостави инфраструктура за изброените дотук подсистеми от високо ниво, като се грижи за създаването на менюта, раздели, панели и позволи на другите подсистеми да обновяват съдържанието им и да регистрират действия. Тези действия се изпълняват автоматично, когато GUI подсистемата обработи определен тип потребителски вход, който пристига под формата на window събития от операционната система (кликване, влачене, селектиране с мишката, вход от клавиатурата и други).

**Връзка с други компоненти и особености:** Подсистемата за нотация е система от междинния слой и зависи от тези от по-ниския слой – те й предоставят функционалности за интернационализация, персистентност, конфигурационни опции, logging и така нататък. Характерно за нея е, че тя не притежава знание за и съответно не зависи от подсистемите за модела, профилите, критиците, диаграмите и други, имплементирана е директно върху Java и Swing. За сметка на това тя е в директна комуникация както с потребителя, от който получава вход чрез събития, така и с подсистемите за преглед, диаграми, панели със свойства и нотация – те регистрират действия, които тя активира при настъпване на посочените за тях условия. Освен това може да приема обновления от подсистемата за менажиране на задачи, за да обнови съответната визуализация на прогреса на работещите в момента задачи.

**API и имплементация:** Тази подсистема предоставя средства за изграждане на модерен потребителски интерфейс – менюта с действия и менюта с инструменти, средства за създаване на панели и раздели за настройки, без да се интересува от семантиката им. Повечето от тези класове са стандартни Swing класове или техни директни разширения. Тази функционалност може да се достъпи директно от фасадния клас – org.argouml.ui.GUI. Като изключим това, другият важен елемент на тази подсистема е т.нар. TargetManager. Това е клас, който следи текущата потребителска селекция, и предоставя тази информация на другите подсистеми, за да работят коректно панелите със свойства, системата за преглед и други. Освен това той съхранява история на селекциите и предоставя методи navigateBack и navigateForward, което е голямо удобство за потребителя.

### подсистема за зареждане на модули

**Местоположение**: Подсистемата за зареждане на модули се намира в пакета org.argouml.moduleloader

**Отговорности**: Тази подсистема има задачата да осъществява зареждане, регистриране и деинициализацията на модулите на ArgoUML. Грижи се както за модулите на ArgoUML, така и за външни модули и плъгини.

**Връзка с други компоненти и особености:** Подсистемата за зареждане на модули е система от междинния слой и зависи от тези от по-ниския слой – те й предоставят функционалности за конфигурационни опции, logging и така нататък. Тя не комуникира директно с други подсистеми, за да върши работата си, но от гледна точка на другите подсистеми е изключително важна, тъй като се грижи за зареждането и след това за деинициализацията им.

**API и имплементация:** Тази подсистема използва Loadable Proxy Pattern, като основният клас е Module Loader. Той сканира наличните в директорията ext jar-ове и ги използвайки техните манифести, зарежда посочените класове. Това е възможно, стига в модула да има клас, имплементарищ ModuleInterface. След това е възможно динамично зареждане и пре-зареждане на модули.

## Слой от най-високо ниво – подсистема приложение (application)

Тук има само една подсистема – подсистема за приложението. Тя се грижи за инциализацията на всички останали подсистеми, и поради това зависи от всички тях, но никоя не зависи от нея. Оттук започва работата на програмата, зарежда се всичко необходимо, включително и текущия проект. Няма да я разглеждаме подробно, тъй като има главна роля единствено при инициализация. Намира се в пакета org.argouml.application.

## Подсистеми, които могат да се зареждат (loadable modules, plugins)

Това са отделен слой от подсистеми, които се свързват с останалите само чрез имплементиране на интерфейс. Те лесно могат да бъдат изключвани и включвани индивидуално, посредством подсистемата за зареждане на модули. Взаимовръзките помежду им и с останалите системи можем да видим на фиг. 32.



**Фиг. 32.***Изглед към зареждаемите модули и плъгините.*

### Подсистеми за генериране и reverse engineering на код

**Местоположение**: Подсистемите за генериране и reverse engineering на код се намират в пакетите org.argouml.language от проекта argouml-language.

**Отговорности**: Тези подсистеми имат три основни функционалности. На първо място позволяват моделът да бъде конвертиран в java код – това е особено полезно, когато работим с класови и компонентни диаграми. След това подсистеми позволяват моделът да бъде обновяван директно или през java кода. Не на последно място, благодарение на тях може от код, написан на ограничено подмножество от Java, да се генерират UML диаграми.

**Връзка с други компоненти и особености:** Подсистемите за генериране и reverse engineering на код са системи, които могат да се зареждат динамично, затова те не зависят на другите подсистеми, освен на системата за зареждане на подсистеми, която трябва да ги зареди. Осъществяват достъп до външни файлове (сорс код файлове) и са свързани чрез системата за събития с потребителския вход. Освен това се регистрират за събития, генерирани от модел и профил подсистемите, за да могат да обновяват генерирания код заедно с модела и обратното.

**API и имплементация:** Имплементацията на генерирането на код от модели разчита главно на CodeGenerator класа, който дефинира правила, по които поддържаните от конверсията диаграми се превеждат до Java код. Далеч по-интересна е системата за reverse engeering, която използва библиотеката Antlr, за да дефинира граматика, която превръща ограничено подмножестно на Java директно в обекти от подсистемата за модела, без да построява цялото абстрактно синтатиктично дърво, за да пести памет. Освен това в тези подсистеми има и множество помощни класове, които се грижат за консистентност между генерирания код и оригиналния модел, представен на диаграмите.

### Подсистема за списъци с предстоящи задачи

**Местоположение**: Подсистемата за генериране на списъци с предстоящи задачи се намира в пакета org.argouml.congitive.checklist

**Отговорности**: Тази подсистема има само една задача – да следи, визуализира и поддържа списъците за предстоящи задачи. Тя е тясно свързана с подсистемата на дизайн критиците.

**Връзка с други компоненти и особености:** Тази подсистема е изключително тясно свързана с подсистемата на дизайн критиците. Те създават и при завършване премахват списъците с предстоящи задачи. Друга връзка е със системата за потребителски вход, която праща събития при вход на потребителя (например зачертава на елемент като завършен).

**API и имплементация:** Кодът е доста прост – Checklist и ChecklistItem изграждат визуализацията на списъка, а CheckManager следи за прогреса по изпълнението му.

### Подсистема за работа с OCL

**Местоположение**: Подсистемата за работа с OCL се намира в пакета org.argouml.ocl

**Отговорности**: Тази подсистема имплементира функционалностите, свързани с поддръжката на OCL в ArgoUML.

**Връзка с други компоненти и особености:** Като подсистема, която се зарежда динамично, тя не зависи от други подсистеми, освен от системата за зареждане на модули, която е инициализира. Има връзки с потребителския вход и с подсистемата на модела – получава събития от event bus-a,

**API и имплементация:** Кодът е доста прост – Checklist и ChecklistItem изграждат визуализацията на списъка, а CheckManager следи за прогреса по изпълнението му.

### Подсистема за критици и помощни инструменти

**Местоположение**: Подсистемата на критиците се намира в няколко пакета: org.argouml.cognitive, org.argouml.uml.cognitive.critics, org.argouml.pattern.cognitive.critics, org.argouml.language.java.cognitive.critics

**Отговорности**: Тази подсистема предоставя помощ на потребителя въз основа на текущия модел, с който работи, като това става основно чрез дизайн критици – класове, които откриват артефакти в модела, които нарушават зададени от потребителя правила или добри практики, и wizards, които предоставят инструкции стъпка по стъпка за справяне с някои от проблемите, открити от критиците.

**Връзка с други компоненти и особености:** Като подсистема, която се зарежда динамично, тя не зависи от други подсистеми, освен от системата за зареждане на модули, която е инициализира. Има връзки с потребителския вход и с подсистемата на модела и профилите, от които получава информация за текущото състояние на модела, въз основа на която прави анализи. Това става с помощта на event bus-a. От своя страна е свързана със системата за списъци с предстоящи задачи – генерира ги и ги унищожава с помощта на директно извикване на методи.

**API и имплементация:** Основни в тази подсистема са именно класовете критици. Те са няколко типа – класовете от пакета org.argouml.cognitive.critics са базови критици, съвсем общи – например един от тях дава препоръки да не се припокриват диаграми от модела физически върху екрана, защото това пречи на четимостта. Вторият тип критици са в пакета org.argouml.cognitive.critics – те проверяват правила, свързани с UML – например предупреждават кога клас има твърде много операции, което би го направил труден за поддръжка. За тези проблеми в същия пакет има и Wizards, които улесняват решаването им. В пакета org.argouml.pattern.cognitive.critics пък са критици, свързани с шаблони за дизайн, например Singleton. В пакета org.argouml.language.java.cognitive.critics имаме критици, които следят за добри практики при разработка на Java – в момента предупреждават, когато се моделира множествено наследяване. Основен клас за т.нар. wizards е org.argouml.kernel.Wizard.

Една от силните страни на тази подсистема е, че лесно може да се разширява. Може да добавим нови критици, като наследим CrUML, като на всеки нов критик може да дадем заглавие, с което го идентифицираме, и описание за потребителя, категория на проблемите, които открива (STORAGE, PATTERN, METHODS), приоритет – LOW, MEDIUM, HIGH, и UML мета-класове, към които се отнася. След това имаме предикат predicate2(Object dm, Designer dsgr), който ще проверява условието, асоциирано с критика – той приема UML същност dm и втори аргумент, неизползван за момента. Върнатият резултат е NO\_PROBLEM или PROBLEM\_FOUND. Може да се използват редица помощни функции, част от подсистемата на критиците.

## Външни подсистеми

Това са всъщност външни библиотеки, които не са разработени като част от ArgoUML. Всички подсистеми на ArgoUML може да зависят от тях, но тук спадат само библиотеки, които засягат повече от една подсистема. Типични примери са самите стандартни библиотеки на JRE, Аntlr – библиотека, която се използва за създаване на граматики за генериране на ко по моделите, GEF – Graph Editing Framework, която се използва при визуализацията, и подсистемата за документиране и трасиране на хода на програмата – използва се log4j, който поддържа записи с различна степен на важност – FATAL, ERROR, WARN, INFO, DEBUG. Всички тях споменахме, когато разглеждахме релевантните подсистеми, които ги използват.

## Deployment

Що се отнася до deployment диаграмата на ArgoUML, тя се оказва изненадващо проста. Причината е, че приложението е монолитно, не функционира в монолитна среда, а е обикновено десктоп приложение. Единственият допълнителен артефакт, който му влияе, е конфигурацията, която се зарежда от файлове и може да се редактира от потребителя. Въпросната диаграма е на фиг. 33.



**Фиг. 33.***Deployment диаграма на оригиналната архитектура*

# Оценка на архитектурата на ArgoUML

## Предимства

### Разделяне на архитектурата на подсистеми

Използването на архитектура, която дефинира подсистеми, прави структурата на програмата унифицирана и позволява добра организация на кода. Всяка подсистема извършва точно дефинирана дейност, което помага за по-лесно откриване и разрешаване на проблеми в имплементацията на софтуера. Подсистемите скриват вътрешната структура и зависимостите между класовете вътре в себе си. Това от своя страна позволява лесно да се промени коренно имплементацията на подсистемата, без да се налага промяна по целия код на софтуера. Когато кодът е разделен на подсистеми, е много лесно да се дефинират дори локални архитектури в подсистемите, които да са пригодени за спецификата им. Комуникацията и зависимостите са ограничени само в подсистемата. Класовете в една подсистема могат да комуникарат само с класове от същата подсистема. Комуникацията между подсистемите се изолира от специфичните им дейности и позволява по-лесно дефиниране на нужните данни, които де се предават. Ако не се използва дизайн, определящ някакво разделение и групиране на имплементацията по определени критерии, това би влошило доста качеството на софтуера. Липсата на подсистеми е предпоставка за добавяне на излишни зависимости по време на изграждането на архитектурата, но не е ограничено само до този момент. Големи проблеми могат да възникнат при разширяване на вече съществуваща функционалност или имплементиране на напълно нова. В такива случаи липсата на ясно обособени единици от по-високо ниво често става причина за разширяване на функционалността на класовете. Така един клас започва да върши коренно различни неща, не може да се преизползва и налага създаването на нов, който да дуплицира част от неговата дейност. Тази дупликация на кода прави невъзможна поддръжката му и не позволява по адекватен начин да продължават да се добавят още нови функции. Архитекурите с подсистеми позволяват удобно групиране на кода, улесняват комуникацията, дават възможност за използване на специфични архитектури в подсистемите, лесни са за поддръжка, а също и дефинират добра структура, позволяваща добавяне на нови функции. Това пряко е обвързано с изискванията за лесна разширяемост на архитектурата и е предпоставка и за добра производителност.

### Използване на шаблона за дизайн Model-VIew-Controller(MVC)

Използването шаблона за дизайн MVC тук е особено подходящо, тъй като по естествен начин се разграничават потребителския интерфейс от конструираната диаграма като обект и стоящият зад нея модел или език, в случая UML, който придава семантика на диаграмата, която сама по себе си може да бъде разгледана просто като граф. От една страна, този шаблон е пряко обвързан с качествения атрибут за лесно разширяване възможностите на системата – при присъстващото добро разделение е доста лесно да внесем сериозни промени – подменяйки само слоя на модела, може да поддържаме нови езици и типове диаграми, а променяйки само слоя на изгледите, може да променим съществено визуализацията. От друга, той има не по-малко важна роля за гарантиране на свойството лекота за използване – наличието на отделен слой изгледи позволява лесното добавяне на много такива, които позволяват на потребителя да разглежда модела от различни перспективи и улесняват работата му. Освен това съхраняването на цялата информация за модела на едно място, съчетано със системата на събитията за нотификация, позволява промяната в един изглед бързо да се пренесе до всички други, което в крайна сметка гарантира консистентност, а това е много важно за създаване на усещане на лекота при използване на програмата.

### Поддръжка на динамично зареждащи се модули и плъгини

Наличието на опцията за динамично добавяне на модули и плъгини е безспорен плюс в стремежа да се постигне лесно разширяване на системата. Възможността да се добави нова съществена функционалност, без да трябва да се променят съществуващите системи, е гарант за добър отговор на системата при всякакви нужди за разширавяне на възможностите й. Плъгините са свързани и с качеството лекота на употреба – те предоставят допълнителни възможности за улеснение на потребителя. Конкретно дизайн критиците, списъците с предстоящи задачи, функционалностите за генериране на код по диаграми и обратното позволяват много по-лесната употреба на системата както за начинаещ, така и за напреднал потребител, боравещ със сложни модели, при които невинаги е лесно да се държи винаги в умозрението цялостната картина на проекта.

### Подобрение на качеството с помощта на останалите подсистеми

Не на последно място останалите подсистеми също допринасят за съблюдаването на нефункционалните изисквания. Системата за интернационализация улеснява употребата на системата от по-широк набор от потребители. Системата за logging улеснява поддръжката на системата, помага за откриване на проблеми и по-бърза итерация. Системата за конфигурация има отношение към лесното адаптиране на системата за нуждите на потребителя, което е свързано както и с мярката за лекота на употреба, така и с концепцията за разширяване възможностите на системата. Подсистемата за persistence е основното средство за гарантиране на атрибута надеждност – благодарение на нея се съхраняват направените от потребителя промени, за да не загуби много данни дори при сриване на системата. Магистралата за събитията и подсистемата за менажиране на задачи играят важна роля за изпълняване на условието за интерактивност, тъй като позволяват системата да не стои блокирана при тежки фонови задачи, а да остава способна да отговаря на потребителския вход.

Всички тези положителни характеристики водят до извода, че разделянето на подсистеми, MVC шаблона и текущо наличните подсистеми са резултат от добри дизайн решения и бихме искали да ги запазим. Разглежданата архитектура обаче далеч не е перфектна и следва да разгледаме и нейните недостатъци.

## Недостатъци

Въпреки всички тези предимства конкретната имплементация на ArgoUML има някои недостатъци.

1. Цялата програма използва само един процес с една нишка в него. При възникване на грешки в която и да е подсистема, дори такава, която да не е от първостепенна важност, има голяма вероятност програмата да спре да работи.
2. Няма ясно дефиниран и унифициран интерфейс за комуникация между подсистемите. Това може да доведе до излишно усложняване на комуникацията между тях и да попречи на поддръжката и по-нататъшни опити за разширяване възможностите на системата.
3. Комуникацията се извършва предимно чрез директно извикване на методи от публичния интерфейс. Така лесно се предават изключенията, възникнали в една подсистема, на друга, което директно влияе негативно върху надеждността на системата
4. Външните плъгини работят в нишката на подсистемата, която използват. Грешка в плъгина ще доведе до спиране на цялата програма, което отново е проблем на надеждността.
5. Няма механизъм за създаване на плъгин, който да използва данни от няколко подсистеми. Поради това трудно биха могли да се създадат по-сложни плъгини, което е пречка пред бъдещото разширяване на възможностите на системата.
6. Не се дефинира структура на данните, които се пренасят между подсистемите. Това е предпоставка за излишно прехвърляне на данни и може да влияе негативно върху производителността и интерктивността на системата.

# Алтернативна архитектура

Новата архитектура, която ще предложим е избрана на базата на функционалните изисквания на проекта като се опитва да реши вече представените проблеми на текущата.

## Обща структура

Предложената архитектура използва дизайн, при който ясно да се обособят отделните части на софтуера в зависимост от задачите, които решават. На базата на това разделение ще се изградят модули, като всеки модул се описва с 3 компонента – каква дейност извършва, какъв е форматът на неговия вход и форматът на неговия изход. Архитектура, базирана на модули, е доста по-добра от просто логическо разделение на имплементацията. Важно е да се отбележи, че това не са два коренно различни подхода в дизайна на софтуерните архитектури. Точно обратното, модулният дизайн използва логическото разделение, за да може да дефинира ясно определени физически единици (модули) от по-високо ниво. Докато обектно-ориентираното програмиране дава възможност за скриване на имплементационните детайли, модулите дават възможност за енкапсулиране на вътрешните зависимости между елементите. Входът, който приема модулът, се дефинира от публичен интерфейс, който е общ за всички. Единствената разлика е в типа данни, които приема и предава. Изходът на модулите е условно дефиниран за пълнота и съвпада с входа на модула, на който се предават данните. Класовете, които се включват в модула, могат да комуникират помежду си по всякакви начини, тъй като са независими от останалите модули. Това позволява дефиниране на вътрешни архитектури, които да решават специфични проблеми. Те все пак са ограничени и нямат директна връзка с елементи от други модули. Когато се налага, може да се добавят повече вътрешни директни зависимости, с цел да се подобри ефикасността на преноса на данни вътре в модула.

## Изпълнение на модулите

С цел още по-добра изолация архитектурата предвижда изпълнението на модулите в различни нишки на един процес, а също и стартиране на някои модул в отделен процес. Това разделение и на ниво изпълнение спомага за по-лесната имплементация на изискванията за програмата. Дали даден модул да се стартира в друга нишка или отделен процес, зависи от неговата дейност, размера на потока от информация, който обработва, а и от неговата надеждност. Можем да изведем следните правила:

1. Ако два модула често си комуникират и са тясно свързани, добра идея е да са в отделни нишки на един процес. В случаи, когато много често си предават данни, дори е най-добре дори да са в една нишка, с цел да се премахне забавянето от синхронизацията по време на комуникация. Това е свързано с изискването за интерактивност и производителност.
2. Модулите, при които има голяма вероятност да възникне грешка и да спрат работа, трябва да са в отделни процеси. Така се постига по-добра изолация на възникналите грешки. Това е пряко свързано с изискването за надеждност.

## Обмен на данни

Обменът на данни се дефинира чрез типовете данни и начинът им на пренос. Архитектурата използва интерфейсът CommonData. Този интерфейс служи като базов за всеки дефиниран публичен тип данни. Публичните типове данни се използват за дефиниране на информация, която се предава между модулите.

Преносът на данни се определя от интерфейса TransferInterface. За разлика от текущата архитектура този интерфейс е базов за всички подсистеми, като целта му е да даде възможност за унифициран обмен на данни. От друга страна позволява скриването на метода за предаване на данни, което позволява бъдещо развитие на системата.

Архитектурата предвижда два метода за предаване на информация: чрез предаване на съобщения (Message passing), когато модулите работят в отделни процеси, и чрез споделена памет, когато модулите са в един процес.

Дали един модул работи в различен процес или нишка, определя и по какъв начин той предава и приема данните. Ако модулът работи в друг процес, той ще комуникира с останалите чрез предаване на съобщения. Ако работи в отделна нишка – тогава използва предаване на съобщения за комуникация с модули от други процеси и споделена памет за комуникация с модули в други нишки на същия процес.

## Предаване на събития

За предаване на събития между модулите архитектурата предвижда използване на callback методи, но по по-различен начин от текущата архитектура – крайната цел е лесна разширяемост на системата и по-добра производителност чрез спестяване на ненужен обмен на информация.

За осъществявана на тази функционалност са нужни 2 метода, които се намират в TransferInterface:

* getAvailableEvents() – метод, връщащ списък със събитията, които даден модул може да генерира.
* registerForEvent(Type, Callback) – регистриране на callback метод при възникване на избраното събитие.

## Плъгини

Плъгините в новата архитектура са обикновени модули. Добавянето на функционалност чрез плъгини не се различава по нищо от добавянето на нов модул. Те могат да работят в нова нишка в процеса, в който работи модулът, който разширяват, могат да работят и в отделен процес. Добра идея е да работят в отделни процеси, тъй като при възникване на грешки в тях, програмата трябва да продължи да работи поради изискванията за надеждност.

## Описание на новата архитектура, приложена върху ArgoUML

Предложената от нас нова архитектура е представена на следната компонентна диаграма (фиг. 34):



**Фиг. 34.***C & C диаграма на предложената от нас архитектура*

### Главен модул (Main Module)

Главният модул работи в процеса, в който се стартира програмата. Този модул има за задача да стартира всички останали модули в програмата, като пази техен списък, и при затваряне да освободи ресурсите им. Външните плъгини се стартират от модула, който разширяват. Ако даден плъгин комуникира с няколко модула, той трябва да се стартира от главния модул. Друга задача, която изпълнява главният модул, е да рестартира спрели модули. Това става, когато някой друг модул му изпрати съобщение за неработещ модул в системата.

### Модул за потребителски вход (User Input Module)

Модулът за потребителски вход работи в отделна нишка в главния процес. От него може да се създаде само една инстанция, тоест това е singleton модул. Грижи се за получаването на входа от мишката и клавиатурата. Предава го на модулите, които са се регистрирали за получаване на съобщение при активност от потребителя.

### Модул за менюта (Menu Module)

Работи в отделна нишка в главния процес. Предава информация за типа на текущата диаграма и изменения в настройките и на модула DiagramManager. Грижи се за създаването и унищожаването на негов подмодул работещ в отделна нишка, който се занимава с показването на бутоните за менютата на екрана.

### Модул за управление на диаграми (Diagram Manager Module)

. Работи в отделен процес. Грижи се за налагането на правила по време на изграждането и изменението на диаграмата. Свързва се с модула за вход, за да получава съобщения при активност от потребителя. Дейността на този модул силно зависи от поведението и работата на потребителя, затова има голяма вероятност да се случат грешки в него. Фактът, че е изолиран в отделен процес, повишава стабилността на системата. Този модул изпраща измененията по диаграмите на модула DiagramModel и задава команди за генериране на изходен код на GenerateSource модула.

### Модул за модел на диаграмите (Diagram Model Module)

Държи информация за диаграмата и може да прилага изменения върху нея. Налага изменения при команда от DiagramManager. Тъй като има голяма вероятност да възникнат проблеми по време на работа на този модул, той работи в отделен процес. Предава данните за диаграмата на модула DiagramView, който ги показва на екрана. Може да получава заявки за информация от диаграмата, която държи, от модула GenerateSource и други модули или плъгини.

### Модул за изглед на диаграмите (Diagram View Module)

Получава данни за диаграмата от модула DiagramModel генерира изходните изображения и ги показва на екрана. Работи в отделен процес. При възникване на проблем по време на генерирането на изображение, програмата като цяло ще продължи да работи, като главният модул в главния процес ще рестартира този модул. След рестартирането си, той може да получи отново данните за диаграмата от модула DiagramModel, което позволява бързо възстановяване на работата и създава впечатление за надеждност на системата у потребителя.

### Модул за генериране на програмен код (Generate Source Module)

Грижи се за генерирането на код от дадена диаграма. Работи в отделен процес. Свързан е с модула DiagramModel, от който получава данните за диаграмата.

Командата за генериране на изходен код получава директно от модула за вход от потребителя. За записване на генерирания код на диска използва IO модула.

### Модул за достъп до файловата система (IO Module)

Работи в отделна нишка на главния процес. Изпълнява заявки за писане и четене на диска.

### Други модули и плъгини

Благодарение на лесното добавяне на допълнителни модули и плъгини системата може да бъде разширявана допълнително. Могат да бъдат добавени дизайн критиците, системите за интернационализация и персистентност и редица други ценни функционалности от оригиналната архитектура. В този смисъл нашето предсложение е по-скоро нейно разширение. Възниква въпросът с какво и дали е по-добро?

## Предимства на новата архитектура

Една от целите на новата архитектура е да позволи лесно справяне с грешки по време на изпълнение на програмата. Стартирането на модулите в различни нишки и процеси помага за решаването на няколко проблема:

1. Забавянето на изпълнението на кода в един модул да не забавя цялата програма. Изпълнението на модулите асинхронно в отделни нишки или процеси позволява запазване на интерактивността дори в случаи на забавяне на изпълнението на кода в някоя подсистема. Така поддържаме гаранциите, необходими за качествения атрибут интерактивност.
2. Спирането на работа на модул ще даде възможност на програмата да се възстанови, като го рестартира.
3. Рестартирането на модул първо не винаги е възможно, а и понякога не решава проблема с възникналата грешка. Фактът, че модулите работят в отделни нишки или процеси, допълнително подобрява стабилността на програмата. При стария подход с извикване на методи и изпълнение на кода в една нишка, ако възникне грешка в определен модул, проблемът ще се пренесе в останалите, зависими от него модули. Новата архитектура позволява грешката да остане в модула, в който е възникнала, а на другите модули да се предаде само информация за проблема.

## Сравнение със старата архитектура

Предложената нова архитектура запазва MVC модела на оригиналната, като спомага за изграждането на по-надежден и удобен за разработка софтуер. По-голямата надеждност е постигната чрез допълнителна изолация на кода, при която изпълнението на модулите става в отделни процеси и нишки. Налага се модулната структура по-ефективно от оригиналната архитектура, като се задават задължителни ограничения за публичните интерфейси на модулите. Добавят се изисквания за типовете данни, които се предават между модулите, с цел да намали излишните трансфери и подобри производителността на софтуера. Така основните качествени атрибути, които подобряваме, са надеждността и производителността, без да губим лесната употреба и лесното разширяване, които бяха предимствата на предишната архитектура. Това, разбира се, има своята цена – новата архитектура е по-сложна, по-сложен става процесът по deployment на продукта и има опасност да се окаже по-трудна за поддържане. Това може да забележим на deployment диаграмата на новата архитектура на фиг. 35:



**Фиг. 35.***Deployment диаграма на предложената алтернативна архитектура*

# Заключение и оценка

Като всеки софтуерен продукт, ArgoUML има своите плюсове и минуси. Положителните му страни не са малко: потребителският интерфейс е интуитивен и удобен, приложението е многоплатформено (благодарение на това, че е избрана Java платформата). За разработчиците голям плюс е пълният достъп до изходния код, което позволява бързо да се локализират и поправят бъгове, а също така и да се добави функционалност, ако такава е необходима. За обикновения потребител безспорен плюс са възможностите за генериране на програмен код по диаграма, които ускоряват процеса на разработка, и дизайнерските критики – своевременното откриване на проблеми в дизайна и даване на предложения за подобрения води до цялостно увеличение на производителността. Към това трябва да добавим и предимствата на архитектурата му – на първо място лекота на употреба и лесно разширяване, съчетани с приемлива производителност и степен на интерактивност.

Разбира се, съществуват и недостатъци. Все още няма пълна поддръжка за UML 2.0 и като цяло новите стандарти бавно навлизат в приложението. Освен това поради недостиг на разработчици се забелязва забавен темп на развитие – последната официална версия е от 2011 година. Архитекурата също има своите недостатъци, най-вече по отношение на надеждността и възможността да възникнат проблеми с производителността, които да попречат на интерактивността. Поради тази причина предложихме алтернативна архитектура, която може да се справи с тези пропуски, но, разбира се, и тя има своите недостатъци в други аспекти.

Отчитайки плюсовете и минусите, нашето мнение е, че ArgoUML все пак е добър избор като средство за дизайн с помощта на UML. Той предлага много удобство и е добра безплатна и отворена алтернатива на платените инструменти. Архитектурата му има много силни страни, а фактът, че е с отворен код, позволява да приложим всякакви желани от нас промени, включително и алтернативни архитектури, които бихме искали да тестваме. Затова цялостната ни оценка от работата с продукта е положителна.

# Библиография

Основен източник:

[AGUML] http://argouml.tigris.org/ (последно използвана на 01.2017)

Други източници:

[ARGRQ] http://argouml.tigris.org/wiki/ArgoUML\_requirements (последно използвана на 01.2017)

[AUDSN] http://argouml.tigris.org/wiki/Design (последно използвана на 01.2017)

[ARGWK] http://argouml.tigris.org/wiki (последно използвана на 01.2017)