Министерство образования и науки Украины
Севастопольский государственный технический университет

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

«ПРОЦЕДУРА СИСТЕМНОГО АНАЛИЗА ПРИ ПРОЕКТИРОВАНИИ ПРОГРАММНЫХ СИСТЕМ »

для студентов-дипломников
дневной и заочной формы обучения
специальности 7.091501

Севастополь 2006

УДК

„Процедура системного анализа при проектировании программных систем” для студентов-дипломников дневной и заочной формы обучения специальности 7.091501 / Сост. Сергеев Г.Г., Скатков А.В., Мащенко Е.Н. – Севастополь: Изд-во СевНТУ, 2005. – с.

Целью методических указаний является оказание помощи студентам в проектировании сложных программных систем и оформления раздела «Системный анализ» пояснительной записки к дипломному проекту.

Методические рекомендации рассмотрены и утверждены на заседании кафедры (протокол № от 200_ г.)

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензенты:
1.
2.

СОДЕЖАНИЕ

ВВЕДЕНИЕ

Как показывает вся история технического прогресса, создание сложных технических систем и сооружений происходит, как правило, на основе двух принципиально различающихся подходов:

- "нисходящее" проектирование (или проектирование "сверху-вниз", или "декомпозиционное" проектирование), связанное с реализацией "с нуля" некоторого глобального инновационного проекта в рамках единой концептуальной идеи (например, так создавались первые образцы ракетно-космической техники, атомные подводные лодки и др.);

- "восходящее" проектирование (или проектирование "снизу-вверх", или "композиционное" проектирование), основанное на модернизации ранее созданных (унаследованных) технических систем и проектных решений и их агрегировании в виде компонентов в качественно иную, а значит, гораздо более сложную систему (например, создание кластерных архитектур из уже функционирующих автономных серверных платформ).

На практике создавать сложные программные системы (СПС) чаще всего приходится именно путем реконструкции унаследованных систем с последующей интеграцией фрагментов, ранее существовавших программных продуктов, т.е. применять второй подход из вышеупомянутых - "композиционное" проектирование. Действительно, характерная особенность СПС - это большое количество унаследованных проектных решений как в смысле общесистемного и прикладного программного обеспечения, так и в смысле ранее накопленной информации в виде баз и банков данных. Это весьма серьезный фактор, который обязательно учитывается при разработке концептуальных решений по архитектуре будущей СПС.

С другой стороны, на основании системного и структурного анализа данной проблемы можно сделать вывод о том, что не менее важным требованием является достижение синергетического эффекта - проектируемая СПС должна обладать заданными в ТЗ свойствами, которыми порознь не обладают входящие в ее состав локальные подсистемы. Реализовать это требование возможно в рамках концепции нисходящего проектирования на основе декомпозиции стратегических целей создания СПС и общих технических требований к ней как единой системе. В результате подобных преобразований удается осуществить переход от общего ТЗ на систему к частным ТЗ на ее составляющие, что определяет оптимальные пути реконструкции унаследованных локальных систем. Единство двух рассмотренных подходов к созданию СПС - важная черта системного проектирования и залог последующего эффективного управления синтезируемой СПС при ее эксплуатации.

В свете вышеизложенного получившие распространение подходы к проектированию СПС на основе преобразования формальных описаний (моделей) с помощью соответствующих CASE-средств представляются не всегда продук-

тивными, поскольку они ориентированы, в основном, на функциональный инжиниринг по принципу "как надо". Это вполне приемлемо в отношении синтеза прикладной логики и заново формируемых под нее структур данных. Однако, вопрос об общесистемном реинжиниринге, т.е. конкретных направлениях реконструкции тех или иных унаследованных компонентов, о путях встраивания этих компонентов в СПС остается открытым. Кроме того, современные CASE-технологии не отражают в должной мере иерархический характер принятия решений и соответствующую этому характеру многоуровневую оргструктуру управления процессами, связанную с выработкой координирующих действий со стороны ЛПР.

Целью настоящих методических указаний является оказание помощи студенту-дипломнику на этапе разработки концептуального проекта сложной программной системы и оформление соответствующего раздела «Системный анализ проекта».

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

1.1 Принципы системного анализа

Принципы системного анализа – это некоторые положения общего характера, являющиеся обобщением опыта работы человека со сложными системами. Различные авторы излагают принципы с определенными отличиями, поскольку общепринятых формулировок на настоящее время нет. Однако, так или иначе все формулировки описывают одни и те же понятия.

Наиболее часто, к системным причисляют следующие принципы: принцип конечной цели, принцип измерения, принцип единства, принцип связности, принцип модульности, принцип иерархии, принцип функциональности, принцип развития, принцип сочетания централизации и децентрализации, принцип учета неопределенности и случайностей. Пренебрежение этими принципами при проектировании любой нетривиальной технической системы, в том числе и программной, непременно приводит к потерям того или иного характера, от увеличения затрат в процессе проектирования до снижения качества и эффективности конечного продукта.

1) *Принцип конечной цели* – основополагающий принцип системного анализа. Конечная цель имеет абсолютный приоритет, в системе все должно быть подчинено достижению конечной цели. Принцип имеет несколько правил:

– для проведения системного анализа необходимо в первую очередь сформулировать цель исследования; расплывчатые, не полностью определенные цели влекут за собой неверные выводы (в этом смысле весьма красноречива расхожая фраза: «Четко сформулированная задача есть уже половина ее решения»);

– анализ следует вести на базе первоочередного уяснения основной цели (функции, основного назначения) исследуемой системы, что позволит определить ее основные существенные свойства, показатели качества и критерии оценки;

– при синтезе систем любая попытка изменения или совершенствования должна быть в первую очередь рассмотрена с позиции его полезности в достижении конечной цели;

– цель функционирования искусственной системы задается, как правило, системой, в которой исследуемая система является составной частью.

2) *Принцип единства*. Система должна рассматриваться и как целое, и как совокупность элементов. Расчленение системы (например, на подсистемы) необходимо производить, сохраняя целостное представление о системе. Принцип подразумевает выделение подсистем, композиция которых в совокупности со связями позволяет выполнять все функции проектируемой системы, определяет ее структуру и может быть рассмотрена как единая, целостная система.

3) *Принцип связности*. Любая часть системы должна рассматриваться со всеми своими связями с окружающими ее объектами, как внешними по отношению ко всей системе в целом, так и внутренними – другими элементами системы. Если некоторая подсистема имеет связи только с внешней средой, то есть смысл реализовать ее в виде отдельной системы. Подсистема, не связанная ни внешней средой, ни с другой подсистемой, является избыточной и должна быть удалена из системы.

4) *Принцип модульности.* В соответствии с этим принципом, выделение модулей системы, если таковые имеются, продуктивно и оправдано. Под модулями здесь понимаются относительно автономные и достаточно простые блоки, выполняющие ограниченный набор функций. Модуль в отличие от подсистем, которые имеют нерегулярную структуру и, как правило, несут определенную функцию, частично отражающую функцию системы, имеет регулярную структуру и характерные для него внутренние и внешние связи. Принцип указывает на возможность вместо части системы исследовать совокупность ее входных и выходных воздействий (абстрагирование от излишней детализации). Выделению модулей соответствует декопмозиция сложной задачи на множество более простых подзадач.

5) *Принцип иерархии*. Продуктивно выделять в системе иерархии, если они существуют. Иерархия [от гр. священная власть – порядок подчинения составных нижестоящих элементов и свойств вышестоящим по строго определенным ступеням (иерархическая лестница) и переход от низшего уровня к высшему] есть тип структурных отношений в сложных многоуровневых системах, характеризуемых упорядоченностью, организованностью взаимодействий между отдельными уровнями по вертикали. Иерархические отношения имеют место во многих системах, для которых характерна как структурная, так и функциональная дифференциация, т. е. способность к реализации определенного круга функций. Причем на более высоких уровнях осуществляются функции интегра-

ции, согласования. Необходимость иерархического построения сложных систем обусловлена тем, что управление в них связано с переработкой и использованием больших массивов информации, причем на нижележащих уровнях используется более детальная и конкретная информация, охватывающая лишь отдельные аспекты функционирования системы, а на более высокие уровни поступает обобщенная информация, характеризующая условия функционирования всей системы, и принимаются решения относительно системы в целом. В реальных системах иерархическая структура никогда не бывает абсолютно жесткой в силу того, что иерархия сочетается с большей или меньшей автономией нижележащих уровней по отношению к вышележащим, и в управлении используются присущие каждому уровню возможности самоорганизации.[1]

6) *Принцип функциональности*. Принцип определяет первичность функции по отношению к структуре, так же, как цель первична для функции. Другими словами, цель определяет функции системы, а функции определяют ее структуру – совокупность элементов с их связями. Структура же, в свою очередь, определяет параметры системы. В случае придания системе новых функций полезно пересматривать ее структуру, а не пытаться втиснуть новую функцию в старую схему. Поскольку выполняемые функции составляют процессы, то целесообразно рассматривать отдельно процессы, функции, структуры. В свою очередь, процессы сводятся к анализу потоков различных видов: материальный поток, поток энергии, поток информации, смена состояний. С этой точки зрения структура есть множество ограничений на потоки в пространстве и во времени.

7) *Принцип развития*. Это учет изменяемости системы, ее способности к развитию, адаптации, расширению, замене частей, накапливанию информации. В основу синтезируемой системы требуется закладывать возможность развития, наращивания, усовершенствования. Обычно расширение функций предусматривается за счет обеспечения возможности включения новых модулей, совместимых с уже имеющимися. С другой стороны, при анализе принцип развития ориентирует на необходимость учета предыстории развития системы и тенденций, имеющихся в настоящее время, для вскрытия закономерностей ее функционирования.

Одним из способов учета этого принципа разработчиками является рассмотрение системы относительно ее жизненного цикла. Условными фазами жизненного цикла ИС являются проектирование, изготовление, ввод в эксплуатацию, эксплуатация, наращивание возможностей (модернизация), вывод из эксплуатации (замена), уничтожение.

Отдельные авторы этот принцип называют принципом изменения (историчности) или открытости. Для того чтобы система функционировала, она должна изменяться, взаимодействовать со средой. На этом принципе основаны саморазвивающиеся и самоорганизующиеся системы.

8) *Принцип сочетания централизации и децентрализации.* Это сочетание в сложных системах централизованного и децентрализованного управления, ко-

торое, как правило, заключается в том, что степень централизации должна быть минимальной, обеспечивающей выполнение поставленной цели.

Недостаток децентрализованного управления – увеличение времени адаптации системы. Он существенно влияет на функционирование системы в быстро меняющихся средах. То, что в централизованных системах можно сделать за короткое время, в децентрализованной системе будет осуществляться весьма медленно.

Недостатком централизованного управления является сложность управления из-за огромного потока информации, подлежащей переработке в старшей системе управления. Поэтому в сложной системе обычно присутствуют два уровня управления. В медленно меняющейся обстановке децентрализованная часть системы успешно справляется с адаптацией поведения системы к среде и с достижением глобальной цели системы за счет оперативного управления, а при резких изменениях среды осуществляется централизованное управление по переводу системы в новое состояние.

9) *Принцип учета неопределенности и случайностей.* Принцип утверждает, что можно иметь дело с системой, в которой структура, функционирование или внешние воздействия не полностью определены.

Сложные открытые системы не подчиняются вероятностным законам. В таких системах можно оценивать «наихудшие» ситуации и рассмотрение проводить для них. Этот способ обычно называют методом гарантируемого результата. Он применим, когда неопределенность не описывается аппаратом теории вероятностей.

При наличии информации о вероятностных характеристиках случайностей (математическое ожидание, дисперсия и т.д.) можно определять вероятностные характеристики выходов в системе. В случае недостатка информации о поведении системы и взаимодействии ее с внешней средой следует предусматривать при функционировании системы реализацию способов пополнения информации о них

Принцип измерения. Система никогда не может быть объективно оценена средствами самой системы. О качестве функционирования какой-либо системы можно судить только применительно к системе более высокого порядка. Другими словами, для определения эффективности функционирования системы надо представить ее как часть более общей и проводить оценку внешних свойств исследуемой системы относительно целей и задач суперсистемы.

Перечисленные принципы обладают очень высокой степенью общности. Для непосредственного применения исследователь должен наполнить их конкретным содержанием применительно к предмету исследования. Такая интерпретация может привести к обоснованному выводу о незначимости какого-либо принципа. Однако знание и учет принципов позволяют лучше увидеть существенные стороны решаемой проблемы, учесть весь комплекс взаимосвязей, обеспечить системную интеграцию.

1.2 Интерпретация принципов системного анализа при проектировании программных продуктов

1) Принцип конечной цели

Как известно, проектирование прикладной программной системы начинается с анализа требований, которым она должна будет удовлетворять. Такой анализ проводится с целью понять назначение и условия эксплуатации системы настолько, чтобы суметь составить ее предварительный проект.

Моделью системы (или какого-либо другого объекта или явления) мы называем формальное описание системы, в котором выделены основные объекты, составляющие систему, и отношения между этими объектами. Построение моделей - широко распространенный способ изучения сложных объектов и явлений. В модели опущены многочисленные детали, усложняющие понимание. Моделирование широко распространено и в науке, и в технике.

Модели помогают:

- проверить работоспособность разрабатываемой системы на ранних этапах ее разработки;
- общаться с заказчиком системы, уточняя его требования к системе;
- вносить (в случае необходимости) изменения в проект системы (как в начале ее проектирования, так и на других фазах ее жизненного цикла).

В основе метода системного анализа лежит принцип «черного ящика» (рис. 1). Его выходы определяются входами и внутренним строением (состянием). В этом смысле говорят, что выход есть функция от входа и самого «черного ящика» Y=F(X,Z).

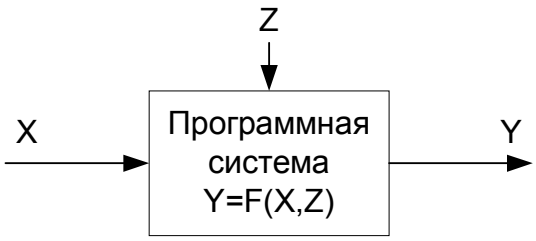

Рисунок 1 – Системотехническое представление программной системы в виде «черного ящика»

Входными данными системы является вектор X, управляющими параметрами - вектор Z (внутреннее состояние программной среды, настройки пользователя, содержимое базы данных и т.п.). Выходными данными системы служит вектор Y.

В соответствии с данным принципом должна быть четко сформулирована конечная цель – назначение проектируемой системы. На основании четко сформулированного назначения можно выделить функции системы, входные и выходные данные системы.

Таким образом, результатом применения принципа конечной цели явля-

ются список функций, которые должна выполнять система, а также спецификация входных и выходных данных в соответствии с ГОСТ 19.202-78.

2) Принцип единства

Согласно принципу единства, в программной системе выделяются подсистемы, каждая из которых выполняет полностью или частично некоторые функции проектируемой системы. Совокупность этих подсистем выполняет все функции системы. Отдельные подсистемы имеет смысл выделять для решения семантически слабо зависимых и достаточно крупных подзадач. В работах [2,3] не рекомендуется выделять слишком много подсистем, оптимальным можно считать выделение 3-7 подсистем. При выборе конкретного варианта декомпозиции проектируемой системы на подсистемы можно использовать один или несколько критериев (метрик) качества системы.

Результатом применения этого принципа является декомпозиция программной системы на подсистемы (модули, формы, библиотеки и т.п.).

3) Принцип связности

Определяются связи между подсистемами – смысловая нагрузка и направления передачи данных, которыми будут обмениваться выделенные подсистемы, а также подсистемы, которые будут обмениваться информацией непосредственно с внешней средой, т.е. принимать входные данные системы и формировать выходные.

4) Принцип модульности

Модульность – фундаментальный аспект всех успешно работающих крупных систем [4].

С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является ее разбиение на более простые и обозримые части. Наиболее широко используемой технологией разработки программных систем на сегодняшний день является объектно-ориентированный подход к программированию (ООП). Объектно-ориентированный подход основан на систематическом использовании моделей для языково-независимой разработки программной системы.

Выделение *класса* является естественным развитием модульности. В классе структуры данных и функции их обработки объединяются. Класс используется только через интерфейс – детали реализации для пользователя класса несущественны. Идея классов отражает строение объектов реального мира – ведь каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами, свойствами и поведением. Программы часто предназначены для моделирования предметов, процессов и явлений реального мира, и классы являются удобным и адекватным инструментом для представле-

ния моделей в языке программирования.

Конкретные величины типа «класс» называются *экземплярами класса*, или *объектами*. Объекты взаимодействуют между собой, посылая и получая сообщения. Сообщение – это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций. Таким образом, с помощью ООП легко реализуется так называемая «событийно-управляемая модель», когда данные активны и управляют вызовом того или иного фрагмента программного кода.

Класс содержит данные и функции для их обработки. Другим классам нежелательно иметь собственные средства обработки этих данных, они должны пользоваться для этого функциями первого класса. Для того чтобы использовать класс, нужно знать только его интерфейс, а не все детали его реализации. Чем более независимы модули, тем легче отлаживать программу. Скрытие деталей реализации называется *инкапсуляцией*. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Интерфейсом модуля являются заголовки всех функций и описания доступных извне типов, переменных и констант.

В ООП идея инкапсуляции получила свое логическое завершение. Инкапсуляция повышает степень абстракции программы: данные класса и реализация его функций находятся ниже уровня абстракции, и для написания программы информация о них не требуется. Кроме того, инкапсуляция позволяет изменить реализацию класса без модификации основной программы, если интерфейс остался прежним [2]. Реализации отдельных классов также рекомендуется помещать в отдельные модули.

Следующим шагом в повышении уровня абстракции программы является группировка классов в отдельные файлы (модули), компилируемые раздельно. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля, позволяя отлаживать программу по частям (или разными программистами).

Результатом применения принципа модульности является декомпозиция подсистем и системы в целом на головную программу, модули, библиотеки.

5) Принцип иерархии

Построение иерархий, согласно принципам системного подхода к анализу задачи, может оказать существенную помощь в процессе формирования СПС.

Принцип иерархии в проекции на процесс разработки программных систем реализуется двумя базовыми свойствами ООП: наследованием и полиморфизмом.

Наследование – это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы. Выделение общих черт различных классов в один класс-

предок является мощным механизмом абстракции.

Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях и листьях.

Полиморфизм – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

Разработка иерархии классов является нетривиальной задачей. Грамотно спроектированные иерархии классов позволяют создавать высокоэффективные, легко читаемые и понимаемые программы. Плохо спроектированная иерархия приводит к созданию сложных и запутанных программ.

Важно до начала проектирования правильно определить, требуется ли вообще применять объектно-ориентированный подход. Если в иерархии классов нет необходимости, то, как правило, достаточно ограничиться модульной технологией [2].

Формальные признаки ситуации, когда выделение группы объектов в иерархию классов оправдано:

- наличие семантической связи между объектами;
- общие для объектов свойства, которые могут быть вынесены в базовый класс;
- общие для объектов методы – функции обработки данных, инкапсулированных в классе;
- различные в реализации, но семантически эквивалентные функции различных объектов могут быть включены в иерархию механизмов на базе полиморфизма, в частности, с помощью виртуальных методов, при этом в базовом классе может присутствовать только декларация метода, а реализация ложится на его потомков – такие базовые классы называются *абстрактными*.

Таким образом, с помощью механизмов ООП возможно объединение в иерархию объектов, различных по своей природе, но схожих по сути, т.е. на абстрактном уровне. Например, можно выделить в отдельную иерархию следующие объекты: файл, процесс, устройство и область памяти; определить для них базовый абстрактный класс, содержащий декларации методы чтения, записи и т.п., а в производных классах объектов соответствующим образом их реализовать. Это позволит формализовать обращение к этим объектам и оперировать ими одинаковым образом, абстрагируясь от их реальной природы.

Результатом применения принципа иерархии являются иерархии классов.

6) Принцип функциональности

На основании назначения подсистем и связей между ними для каждой из них определяются соответствующие им входные и выходные данные и функции.

Результатом применения принципов единства, связности и функциональ-

ности является структура (или множество возможных структур – тогда конечный вариант выбирается на этапе вариантного анализа) проектируемой системы и соответствующая ей структурная схема.

Основным требованием к синтезированной структуре является покрытие объединением множеств функций подсистем множества функций проектируемой системы, определенного применением принципа конечной цели.

7) Принцип развития

Принцип подразумевает учет возможности дальнейшего развития проектируемого программного продукта за счет:

- расширения предметной области;
- расширение функциональности;
- увеличения адекватности используемых моделей, которое влечет за собой возрастание интенсивности потоков обрабатываемой информации;
- переноса программного продукта на другие платформы;
- расширения и модернизации путем замены старых или добавления новых модулей, к примеру, на основе плагин-технологий;
- накопления информации о функционировании системы и ее анализа и т.п.

8) Принцип сочетания централизации и децентрализации

Примером применения данного принципа может служить разделение пользовательского интерфейса управления системой на различные секции, разграничение прав доступа при администрировании системы для различных групп и отдельных пользователей, способ реакции системы на некоторое воздействие или событие (например, возможность напрямую обращаться к аппаратуре или жесткое регламентирование оного операционной системой через обращение к драйверу соответствующего устройства) и т.п.

9) Принцип учета неопределенности и случайностей

После выполнения данного принципа проектируемая система должна предусматривать определенную реакцию на неопределенные ситуации. Должны учитываться способы обработки некорректных входных данных, исключительных и аномальных ситуаций.

1.3 Оценка качества программных систем

Процессы разработки, приобретения и внедрения сложных систем, к которым относятся в частности программные комплексы, должны находится под жестким управленческим контролем. В настоящее время практически во всех организациях обеспечивается контроль важнейших характеристик, связанных с производством и использованием программных продуктов, таких как время,

финансовые средства, ресурсы и т.п. Однако в большинстве случаев вне пределов сферы контроля оказывается наиболее важная характеристика программных продуктов, ради которой, собственно и осуществляются затраты времени, финансовых средств и ресурсов – это качество продукта, поскольку «невозможно контролировать то, что нельзя измерить».

Отсутствие возможности установки полного контроля вызывает рост количества необоснованных решений, увеличивает финансовые и проектные риски, связанные с разработкой и внедрением систем.

Однако в настоящее время уже существуют организации, в которых накоплен достаточно большой опыт использования метрик в управлении качеством разрабатываемых и внедряемых программных продуктов. Использование апробированных подходов в управлении качеством разработки и внедрения крупных программных систем значительно повышает предсказуемость проектов, снижает финансовые и ресурсные издержки.

Среди используемых метрик качества программного обеспечения есть универсальные метрики, которые применимы практически ко всем видам программного обеспечения. В то же время большая часть наиболее важных метрик в успешных проектах разрабатывается индивидуально на основе особенностей проекта и характеристик предметной области.

1.3.1 Определение качества

Сейчас существует несколько определений качества, которые в целом совместимы друг с другом. Приведем наиболее распространенные:

Определение ISO: Качество - это полнота свойств и характеристик продукта, процесса или услуги, которые обеспечивают способность удовлетворять заявленным или подразумеваемым потребностям [3].

Определение IEEE: Качество программного обеспечения - это степень, в которой оно обладает требуемой комбинацией свойств [4].

1.3.2 Идентификация и классификация характеристик качества

Исследования метрического анализа качества показывают [5], что не существует единственной метрики, которая бы дала универсальный рейтинг качества программного обеспечения. Измерения качества дают спектр проектно-зависимых метрик, которые являются руководящей основой для принятия решений в процессе разработки, заказа и сопровождения программного обеспечения.

Следует отметить, что метрики качества являются изначально неочевидной категорией. Исторически сначала были выделены ряд универсальных и неполных метрик на основе следующих шагов:

1. Определение множества характеристик, которые, являясь важными для программного обеспечения, допускают несложное измерение и не перекрываются.

2. Выделение кандидатов в метрики, которые измеряют степень удовлетворения указанным характеристикам.

3. Исследование характеристик и связанных метрик, для определения корреляции, значимости, степени автоматизируемости.

4. Исследование корреляции между метриками, степени перекрытия, зависимости и недостатков.

5. Рафинирование множества метрик в целом во множество метрик, которые в совокупности адекватно отражают качество программного обеспечения.

6. Корректировка каждой метрики в итоговом множестве в контексте зафиксированных множеств характеристик и метрик.

На основе систематического применения данного подхода были выведены примеры универсальных характеристик программного обеспечения, структурно связанные в иерархию). Нижний слой характеристик в иерархии должен быть строго дифференцирован для того, чтобы исключить (или минимизировать) перекрытия. Данный слой должен состоять из примитивных характеристик, допускающих измерение.

Измерение характеристик нижнего слоя может происходить путем ручного сбора информации, специальными автоматизированными средствами, возможен экспертный способ.

Для конкретного проекта должно быть разработано или дополнено свое множество метрик, которое отражает назначение и особенности окружения разрабатываемого программного продукта.

1.3.3 Цена качества

Понятие цены качества первоначально была введена Джураном (J.M. Juran) и Грином (F.M. Gryna) как стоимость в составе продукта, которая может быть сэкономлена, если все исполнители работают безупречно.

Цена качества – важная категория, поскольку фактически она отражает стоимость работ на доработку, увеличенную стоимость сопровождения.

Цена качества может быть разделена на два главных типа: согласованная (conformance) и несогласованная (non-conformance).

Согласованная цена качества - это сумма, затраченная на достижение качества продукта. Она делится на цену предупреждения (Prevention cost) и цену контроля (Appraisal cost). Затраты предупреждения связаны с предупреждением дефектов прежде чем они произойдут. При разработке ПО примером затрат предупреждения являются затраты на обучение коллектива базовым методологиям, переход на современные технологии разработки, использование автоматизированных средств проектирования и разработки.

Цена контроля включают измерение, оценивание или ревизию продукта для обеспечения гарантии соответствия между требованиями, стандартами качества и результатом. Для разработки программного обеспечения, например, затраты оценки включают в себя инспекцию программ, тестирование и измере-

ние программных показателей.

Несогласованная цена качества включает все издержки понесенные, вследствие выявления недостатков, возникновения ошибок и выхода из строя. Несогласованная цена качества включает внутренние и внешние издержки.

Внутренние издержки связаны с проблемами, выявленными до того, как продукт отправлен заказчику. Для разработки программного обеспечения сюда включены затраты на переработку программ, повторную инспекцию и тестирование. Затраты связанные с ошибками, проявившимися при эксплуатации продукта у заказчика, относятся к внешним издержкам. Для программного обеспечения, например, сюда включены затраты на сопровождение и поддержку, убытки от простоев и некорректного функционирования.

Проведенные исследования показывают: чем выше качество процесса разработки, тем выше качество разработанного в этом процессе качества программного обеспечения. Качество на каждой стадии проекта возрастает, во-первых, как прямое следствие зрелости процесса, во-вторых, вследствие использования промежуточного продукта более высокого качества, произведенного на предыдущей стадии. При этом подчеркивается, что значение второй причины обеспечивающей нарастание качества в процессе жизненного цикла для зрелых процессов оказывается гораздо более важным.

Отсюда вытекают следующие следствия:

• Первое: качество накапливается в продукте при сложном производстве кумулятивным образом, причем, вклад в качество, осуществленный на ранних стадиях, имеет более сильное влияние на конечный продукт, чем на более поздних стадиях. Это подтверждается всей практикой программирования, например, известно, что недостатки проектирования систем не могут быть компенсированы высоким качеством кодирования.

• Второе: тестирование и измерение качества должно происходить на всех стадиях жизненного цикла. Извлечение данных о качестве, которое было заложено на ранних стадиях, может быть очень дорогим, при отсутствии полных результатов промежуточного продукта на предыдущих стадиях, например, при отсутствии логического дизайна системы, требований к продукту и т.п.

Таким образом, для управления качеством построения сложной системы необходимо производить выбор производителей на основе измерения степени зрелости и прозрачности используемых процессов разработки. Измерение качества процесса разработки подрядчиков является важной составной частью общего управления качеством, более важным, чем измерение качества результирующего продукта, производимого в ходе приемо-сдаточных испытаний.

1.3.4 Качество программного продукта

Разработка современных больших программных систем в настоящее время все более базируется на компонентной разработке (Component Base System – CBS). Технология построения CBS позволяет значительно снизить стоимость и время разработки. В то же время возрастает риск, связанный с ис-

пользованием в системе программных компонент разработанных различными производителями.

Наиболее действенный способ решения данной проблемы состоит в использовании метрик для управления качеством и рисками при построении CBS, с целью измерения различных факторов влияющих на конечное качество продукта и устранения источников риска [4]. Метрики качества при этом должны быть использованы для обеспечения принятия решений на различных этапах жизненного цикла разработки по экономической целесообразности использования компонент.

Исходные коды компонент, как правило, являются недоступными для конструкторов системы, кроме того, в них предусматривается сложный структурированный интерфейс. Следствием этого является значительное различие между метриками, которые обычно применимы для традиционных систем и метриками для CBS. Большинство традиционных метрик используются на этапе планирования и разработки. Ключевым для управления качеством при использовании метрик в разработке компонентных систем является выбор метрик качества применимых на всех этапах жизненного цикла и оценивающих как качество процесса, так и качество продукта.

Примерами метрик могут служить следующие показатели:

1) Метрики менеджмента:

- Цена (Cost) – расходы на приобретение/разработку
- Время разработки (Time-to-market)
- Среда разработки (Software Engineering Environment)
- Использование системных ресурсов (System Resource Utilization)

Указанные метрики могут использоваться на этапах планирования и контроля проектов и других задач управления или использоваться в качестве параметров управления штатной ERP системы.

Метрика «cost» измеряет общую цену, включая цену анализа рынка, приобретения, интеграции и улучшения качества.

Метрика «time-to-market» - мера времени от формирования заказа на программу до поставки. При итерационной разработке данная метрика модифицируется для измерения времени, требуемого для поставки заданного объема приращения функциональности, то есть скорости поставки.

Метрика «System resource utilization» - определяет процент целевых компьютерных ресурсов, используемых системой.

«Software engineering environment» - мера способности производителя разрабатывать программное обеспечение высокого качества. Данная метрика может быть выражена в терминах модели «Software Acquisition Capability Maturirty Model (SA – CMM) [6]

2) Метрики требований:

- Соответствие требованиям (requirement conformance)
- Стабильность требований (requirement stability)

Метрики требований дают возможность контролировать спецификации,

изменение требований, а также степень их удовлетворения.

3) Метрики качества:

- Адаптируемость (adaptability)
- Сложность интерфейсов и интеграции (complexity of interfaces and integration)
- Тестовое покрытие (test coverage)
- Надежность (reliability)
- Профили ошибок (fault profiles)
- Степень удовлетворения потребностей заказчика (customer satisfaction)

«Adaptability» - мера гибкости системы, оценивает способность системы адаптироваться к изменениям требований либо перепроектированием системы, либо интеграцией приложений.

«Complexity of interfaces and integration» - метрика, измеряющая степень сложности интерфейса или дополнительного программирования требуемого для интеграции компоненты в систему, которые требуются для тестирования, отладки и сопровождения, компенсирующего потерю качества.

Метрики «test coverage» указывают степень полноты различных типов тестирования.

«Reliability»- метрика, оценивающая вероятность работы системы без отказов. Данная метрика может быть получена в рамках традиционного подхода.

«Fault profiles» - метрика, измеряющая кумулятивное число обнаруженных ошибок.

«Customer satisfaction» - метрика, оценивающая степень соответствия программного обеспечения ожиданиям и требованиям заказчика. Данная метрика может быть оценена перед поставкой на этапе опытной эксплуатации на основе прогнозирующих параметров.

Метрики качества, выводимые из требований чрезвычайно важны для анализа качества продукта, однако они создаются на начальных этапах разработки, когда степень неопределенности и риск, связанный с разработкой и внедрением новых программных продуктов велики. Для эволюционного процесса разработки должны быть приняты к рассмотрению метрики качества программ, используемые в процессе реализации циклов разработки.

К числу подобных метрик относится:

- Гибкость (flexability), которая аккумулирует ряд свойств:
- Модульность (Modularity)
- Изменяемость (Changeability)
- Сопровождаемость (Maintainability)
- Адаптивность (adaptability), которая подразумевает:
- Настраиваемость (customizability)
- Переносимость (Portability)
- Способность к взаимодействию (Interoperability)

В ходе приемосдаточных испытаний проходит проверка качества про-

граммного обеспечения, связанного с функциональностью, надежностью, производительностью в соответствии с документами, которые были приняты на начальных этапах проекта. Оценка качества по приведенным выше метрикам, как правило, не проводится. Однако уже через короткое время обычно происходит снижение уровня качества программного обеспечения, связанное с расхождением текущих требований заказчика к системе. Обычно причиной этого является высокая стоимость исправлений или изменений в программной системе.

Исправления программного обеспечения может быть инициировано по следующим причинам:

1) исправление программы с недостаточным уровнем качества (bug fixing);

2) изменение программы для повышения уровня качества (enhancement);

3) изменение программы для удовлетворения изменения в требованиях.

Опыт внедрения и использования крупных программных систем показывает, что стоимость эксплуатации и сопровождения в составе общей стоимости владения системы (total cost ownership - TCO) увеличивается с ростом системы опережающими темпами. Отсюда следует, что показатели качества программ, связанные с гибкостью и настраиваемостью становятся все более важными. Очевидно: чем легче программный продукт модифицировать, тем легче достичь изначальных показателей качества за исключением производительности. Баланс производительности и гибкости – один из ключевых моментов, который должен находится под строгим контролем в критичных областях применения.

1.3.5 Качество технического проекта

Измерение качества проектирования является очень важной составляющей частью в процессе обеспечения качества программного продукта. Особую важность это приобретает при объектно–ориентированном проектировании программных систем. Объектно–ориентированная разработка программ вводит новые факторы качества, связанные с повторным использованием ранее выполненных наработок и выполнением модификаций.

Идеально повторное использование и модификация может быть произведена на нескольких уровнях: уровень исходного кода, физический дизайн, логический дизайн, требования, - и до различной степени: в рамках системы, в рамках проекта, в рамках компании.

Процесс модификации программной системы включает три главных фазы: реструктуризация (создание логически эквивалентной системы); обратный инжиниринг (reverse engineering, анализ системы с целью выделения системных компонент, их функций и взаимосвязей, включая спецификации верхнего уровня системы) и прямой инжиниринг (forward engineering, разработка системы от спецификаций до кодирования и внедрения).

В настоящее время выполняется большое количество проектов, связанных с переводом использующихся унаследованных систем в объектно-ориентированные системы для увеличения их срока жизни и функционального

развития. Критичным в таких проектах является контроль качества, на который ложится задача гарантировать обеспечение роста или неуменьшения уровня потребительских характеристик программных систем [5].

В процессе инжиниринга программных систем в дополнение к классическим метрикам должны быть включены в число наиболее важных такие метрики качества объектно-ориентированного дизайна как: надежность (reliability), сложность (complexity) и возможность повторного использования (reusabiblity).

Измерение качества проектирования является принципиально важной частью процесса разработки, поскольку, как показывает статистика стоимость ошибки проектирования в среднем на два порядка выше стоимости ошибки кодирования [32].

При измерении факторов качества широко используется модель: фактор а критерий а измерение (factor a criteria a measurement). Установка связи фактор а критерий требует анализа составляющих факторов качества. Например:

Надежность – фактор, отражающий возможность установления явных соответствий между требованиями к системе и ее реализацией. Данное соответствие устанавливается через дизайн системы. Игнорирование этого обстоятельства является частой причиной выпуска ненадежных систем.

Сложность – фактор, отражающий трудность понимания, реализации и модификации. Сложные проектные решения трудно реализовывать, при этом они чреваты большими ошибками при реализации.

Возможность повторного использования – фактор, отражающий степень применимости проектных решений в различных контекстах. Наличие таких решений значительно уменьшают размер проекта, повышают выветренность решений и более высокую степень соответствия проблемной области.

В соответствии с анализом, каждому фактору ставятся в соответствие критерии и, далее, метрики для измерения критериев.Далее конкретные метрики выводятся в соответствии с особенностями проекта из критериев качества: accuracy (точность), completeness (полнота), consistency (согласованность), module size (размер модулей), data coupling (связь модулей по данным), cohesion (связность), modularity (модульность), span of control (норма управляемости).

1.3.6 Измерение качества на основе сопровождения продукта

Одной из главных путей способов повышения качества является путь анализа практического опыта использования данного продукта или процесса и использования полученных данных для его совершенствования. Речь идет о так называемой парадигме QIP – Quality Improvement Paradigm [7].

Парадигма совершенствования качества дает систематический подход к организации процессов сбора практического опыта, систематизации накопленной информации и подготовке результирующих выводов для улучшения качества. Одним из главных процессов в дисциплине разработки программ, где этот опыт должен накапливаться, является процесс сопровождения. Потеря значимой информации на этапе сопровождения или несистематический недостаточно

формальный процесс сбора информации неизбежно дает недостаточно объективную или искаженную основу для принятия технических решений о путях развития и совершенствования программной системы

Парадигма улучшения качества представляет собой схему для применения научного подхода в контексте дисциплины инженерии программного обеспечения для совершенствования процесса разработки и связанных с ним результирующих программных продуктов.

Наиболее применимым способом реализации парадигмы улучшения качества является иерархический подход формирования информации: от целей к вытекающим из них вопросам, от вопросов - к метрикам.

Другой частью метода является стандартный научный подход к выработке решений: формулирование проблемы, сбор данных и планирование, формирование гипотез, подготовка к проверке, исполнение проверки, анализ результатов, формулировка решения. Важно подчеркнуть, что данный процесс должен идти строго на метрической основе. В противном случае полезность получаемых результатов значительно снижается.

1.3.7 Методология метрического анализа качества

В настоящее время разработано множество метрик качества программного обеспечения общего назначения, таких как сложность, модульность, тестируемость и т.п. Использование данных метрик, безусловно, очень полезно, вместе с тем следует учитывать, что универсальные метрики в совокупности не дают полного численного описания уровня качества. В частности, вследствие универсальности им присущи следующие недостатки: они базируются в большей степени на лексических и синтаксических характеристиках и не учитывают семантическую составляющую продукта; они сконструированы без учета особенностей требуемой предметной области, операционного окружения и методологии разработки. Традиционные метрики не учитывают качество обрабатываемых данных и взаимодействие данных с потоком вычислений.

Для обеспечения полноты измерения качества требуется на ранних стадиях проекта на основе анализа целей проекта, области применения, ограничений и характеристик разработать проектно-ориентированные (design-oriented) или структурные метрики (structural metrics) качества [5].

Термин «проектно-ориентированный» в данном контексте означает, что метрики разрабатываются в виде стандарта качества проекта на его ранних стадиях и представляют собой правила или нормы (guideline), которым должен удовлетворять промежуточный или конечный продукт. Термин структурный означает, что метрики разрабатываются структурным методом сверху - вниз (top – down) для обеспечения целостности и полноты.

Измерение качества в соответствии с данными метриками состоит в вычислении отклонения фактических характеристик продукта от норм и правил. Методология создания метрик качества указанным способом утверждена в стандарте IEEE 1061 [6]. Схематически методология создания метрик качества

состоит из следующих шагов:

1) Первый шаг (верхний уровень иерархии): Определение нетехнического уровня (то есть уровня предназначенного для менеджеров, пользователей, заказчика):

- Формирование требований качества
- Выбор свойств (атрибутов), установка приоритетов и связи с требованиями
- Присвоение атрибутов факторам качества, которые отражают представление заказчика на качество.
- Установка измерений для факторов качества. Определение допустимых коридоров для величин качества.

2) Второй шаг (средний уровень иерархии): Определение технического уровня (то есть уровня предназначенного для аналитиков, конструкторов, разработчиков). Осуществление декомпозиции факторов качества в измеряемые характеристики программного обеспечения, определяемые как субфакторы.

3) Третий шаг (нижний уровень иерархии). Декомпозиция субфакторов в метрики, которые могут быть применены непосредственно к программному продукту или процессу разработки. Данные метрики служат как непрямые меры (индикаторы) прямых измерений факторов качества на верхних уровнях иерархии. Иными словами это уровень разработанных правил и норм, которым должен удовлетворять продукт или процесс с тем, чтобы были выполнены факторы качества.

Для иллюстрации метода можно привести следующие примеры:

Факторы качества программной системы:

- Переносимость (portability) – усилия, требуемые для переноса системы с одной платформы на другую.
- Надежность (reliability) – ожидаемая степень корректного выполнения системой требуемых функций
- Тестируемость (testability) – усилия, требуемые для тестирования функций программы

Прямые измерения факторов качества:

- Переносимость (portability) – трудоемкость - количество чел.-час., требуемое для переноса программы с платформы X на платформу Y. Допустимый порог: 1 чел.-час. на 1K строк исходного кода .
- Надежность (reliability) – среднее время наработки на отказ. Допустимый порог: 120 операционных дней.
- Тестируемость (testability) – трудоемкость – количество чел.-час., требуемое для полного тестирования 90% всех модулей. Допустимый порог 10 чел.-час. на 1K строк исходного кода.

Следующий шаг – проведение декомпозиции приведенных факторов на субфакторы:

- Точность (accuracy) – правильность вычислений и контроля;
- Сложность (complexity) – трудность разработки и модификации;

- Совместимость (consistency) – использование унифицированной технологии проектирования и разработки на протяжении всего цикла разработки;
- Устойчивость к ошибкам (error tolerance) – степень ущерба от возникающих ошибок;
- Универсальность (generality) – широта потенциального использования;
- Аппаратная независимость (hardware independence) – степень применимости программы на другом аппаратном обеспечении;
- Оснащенность средствами контроля (instrumentation) – степень контроля программы собственного выполнения и идентификации возникающих ошибок;
- Модульность (modularity) – степень функциональной независимости программных компонент;
- Удобочитаемость (readability) – уровень смыслового наполнения комментирования, соответствие стандартам кодирования и именования;
- Простота (simplicity) – легкость понимания программы;
- Системная независимость (system independence) степень независимости от нестандартных характеристик системного окружения и ограничений.

Окончательно, для приведенных субфакторов должны быть определены правила (нормы) и метрики. В случае возможности, целесообразно использовать стандартные метрики для проведения измерений. Приведем несколько примеров:

Сложность (complexity): – использование метрики цикломатической сложности Мак Кааба.

Устойчивость к ошибкам (error tolerance): использование правила (нормы): «все модули должны содержать обработчики предопределенных исключительных ситуаций. Все обрабатываемые исключительные ситуации должны быть либо распространены на внешний уровень, либо разрешены».

Метрики и правила (нормы) могут быть использованы для планирования уровня качества и для непосредственных измерений. Например, метрикой для правила может быть количество нарушений нормы на единицу объема программного обеспечения.

Важным свойством приведенных иерархических метрик является возможность предсказания уровня качества, которое может быть выведено на основе измерения непрямых метрик. При этом получение измерения на верхнем уровне иерархии производится обычно расчетом взвешенной суммы метрик нижнего уровня.

Преимущества методологии качества IEEE основано на возможности обеспечения полноты управления качеством в проектах разработки программного обеспечения.

При этом эффективность управления качеством будет зависеть от тщательности проектирования иерархической структуры метрик качества на основе целей и особенностей конкретного проекта разработки программного обеспечения.

Важной особенностью подхода IEEE является возможность управлять качеством на всех этапах жизненного цикла разработки программного обеспечения, поскольку разработанные правила и нормы (guidelines) для всех факторов качества являются не только метриками, но и инструкциями, выполнение которых может планироваться до исполнения и контролироваться в процессе работы.

Детальная разработка системы качества по данной методике требует времени и средств, поэтому в практике разработки программных систем она используется при создании и внедрении высококритичных программных систем с высокой стоимостью разработки (системы управления транспортом, системы военного назначения, системы обеспечения безопасности, банковские системы и т.п.)

2 ПРИМЕР ПРИМЕНЕНИЯ ПРИНЦИПОВ СИСТЕМНОГО АНАЛИЗА ПРИ ПРОЕКТИРОВАНИИ ПРОГРАММНОЙ СИСТЕМЫ

2.1 Постановка задачи

Разработать программную систему трехзначного моделирования с задержками многоуровневых комбинационных схем на базе элементов И, ИЛИ, НЕ, И-НЕ, ИЛИ-НЕ. Обеспечить следующие возможности системы:

- создание и редактирование, сохранение в файл и последующая загрузка моделей логических элементов и схем;
- добавление и удаление моделей элементов в библиотеку элементов схемы;
- создание и редактирование совокупностей входных наборов для соответствующих им схем (входные сигналы могут принимать значения ‘0’, ‘1’ и ‘x’);
- результаты моделирования должны выдаваться и как конечные значения сигналов на выходах элементов схемы с соответствующими задержками, так и по требованию пользователя в виде временных диаграмм;
- сохранение и просмотр ранее сохраненных результатов.

Описание логического элемента включает:

- идентификатор (например, 2И-НЕ);
- тип (И, ИЛИ, НЕ, И-НЕ, ИЛИ-НЕ, тождественная единица, тождественный нуль);
- количество входов;
- задержки формирования сигналов ‘0’ и ‘1’ на выходе.

Описание схемы включает:

- идентификатор – имя схемы;
- количество входов и выходов;
- список цепей;
- список элементов схемы.

Элемент схемы может быть как логическим элементом, так и другой схемой, и характеризуется:

– идентификатором – функциональным именем;
– типом элемента – идентификатором логического элемента или схемы.

2.2 Системный анализ

Принцип конечной цели

Представим проектируемую программную систему в виде «черного ящика» (рис. 1), тогда входные данные – вектор X – будут включать в себя описание элементов, описание схемы и описание теста. Управляющие параметры системы – вектор Z – это режим выдачи результатов: сокращенная форма (только сигналы на выходах элементов схемы с указанием времени установления) или полная (с временными диаграммами для всех цепей схемы). Выходные данные – вектор Y – это результаты моделирования, представленные в соответствии с режимом их выдачи, а также библиотека моделей элементов схемы, модель схемы и соответствующая ей совокупность входных наборов, сохраненные в один файл, модели элементов, сохраненные в отдельные файлы.

Тогда для выполнения равенства Y=F(X,Z) проектируемая система должна выполнять следующие функции (в совокупности представляющие собой функцию F):

– автоматизированное построение и редактирование моделей логических элементов на основании их описания посредством пользовательского интерфейса;
– автоматизированное построение и редактирование модели многоуровневой комбинационной схемы на основании ее описания посредством пользовательского интерфейса;
– создание совокупности входных наборов для соответствующей схемы;
– сохранение и загрузка из файла определенного формата моделей элементов;
– сохранение и загрузка из файла определенного формата модели схемы, библиотеки ее элементов и совокупности соответствующих ей входных наборов;
– выполнение трехзначного моделирования работы схемы с задержками;
– построение временной диаграммы работы схемы на заданном входном наборе по результатам моделирования;
– просмотр, сохранение и загрузка из файла определенного формата результатов моделирования.

Принцип единства

На основании функций проектируемой системы, представленных выше, в ней можно выделить следующие подсистемы:

– подсистема создания моделей логических элементов;
– подсистема создания моделей схем;

– подсистема организации моделей логических элементов и подсхем схемы – библиотека элементов схемы;
– подсистема задания входных наборов и отображения результатов моделирования;
– подсистема трехзначного моделирования с задержками.

Принцип связности

Совокупность подсистем проектируемой программной системы и их связей – данными, которыми эти подсистемы обмениваются друг с другом и с внешней средой, – образует ее структуру. Структура проектируемой системы представлена на рис. 2.

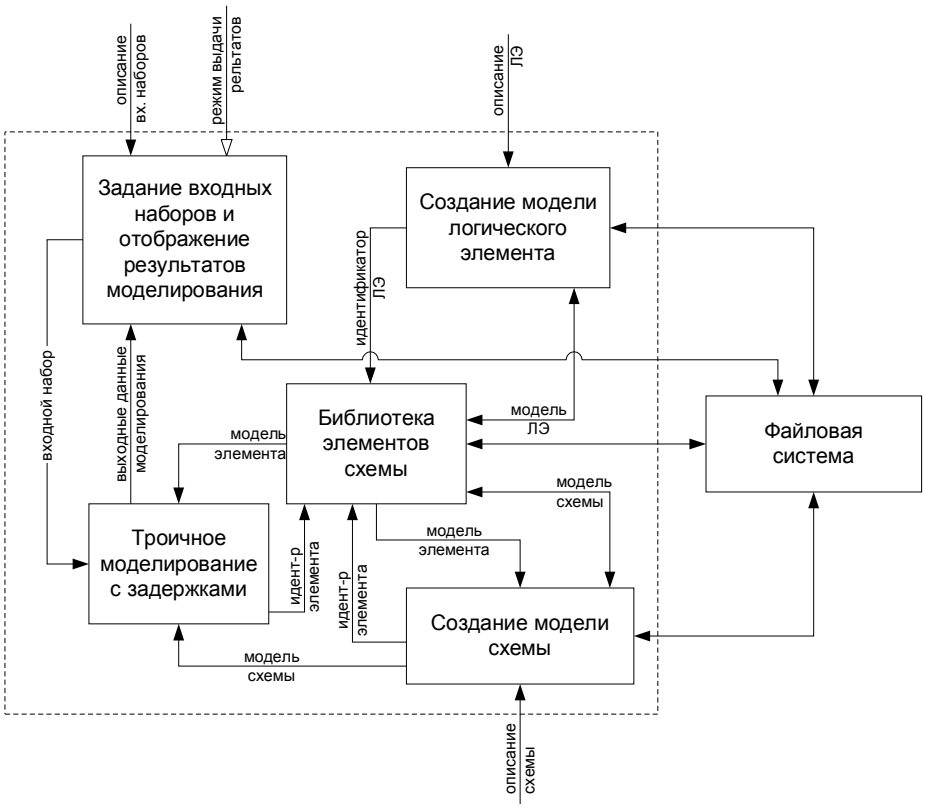

Рисунок 2 – Структура проектируемой системы

Принцип модульности

В проектируемой системе целесообразно выделить следующие модули:
– модуль логических элементов;
– модуль схем;
– модуль библиотеки элементов схемы;
– модуль элементов схемы (реализация классов элементов схемы, соответствующих логическим элементам, подсхемам и входам/выходам схемы как фиктивных логических элементов, для различного их отображения на схеме);

– модули интерфейса оператора (в том числе интерфейс задания входных наборов, интерфейс просмотра временных диаграмм, интерфейс создания модели логического элемента, интерфейс создания модели схемы и т.д.).

Отсутствие отдельного модуля моделирования станет понятным чуть ниже.

Принцип иерархии

Согласно этому принципу в сочетании с методологией объектно-ориентированного программирования, в проектируемой системе можно выделить две иерархии объектов (рис.3 и 4).

Вертикальные связи в совокупности с вершинами (классами) образуют дерево наследования – иерархию программных классов. Методы, выделенные жирным курсивом, являются абстрактными. Все переопределяемые методы являются виртуальными.

На рисунке 3 представлена иерархия классов, соответствующая моделям логических элементов и схем. Пунктирными линиями обозначены семантические связи между элементами и библиотекой элементов схемы.

Класс TBaseElem является базовым абстрактным классом, и определяет основное предназначение всех производных от него классов – выполнять моделирование работы соответствующего ему объекта. Поле ID представляет идентификатор объекта, nIn и nOut – количество входов и выходов соответственно. Метод Simulate выполняет моделирование работы соответствующего объекта, на заданном наборе входных сигналов и возвращает значения выходных сигналов, а также время, за которое эти сигналы установились в свое конечное значение. В данном классе он является абстрактным. Методы Load и Save здесь и далее осуществляют загрузку и сохранение объекта в поток.

Класс TLogicElem соответствует модели абстрактного логического элемента. Поля t01 и t10 определяют задержки перехода сигнала на единственном выходе (nOut=1) элемента из состояния ‘0’ в ‘1’ и из ‘1’ в ‘0’ соответственно (время перехода из состояния ‘x’ в $z \in \{‘0’,’1’\}$ определяется как время перехода из $\bar{z}$ в z).

Классы TAnd, TOr, TNAnd и TNOr реализуют модели логических элементов И, ИЛИ, И-НЕ и ИЛИ-НЕ соответственно (элемент НЕ может быть реализован как элемент И-НЕ или ИЛИ-НЕ с одним входом). Классы соответствующим образом переопределяют метод Simulate.

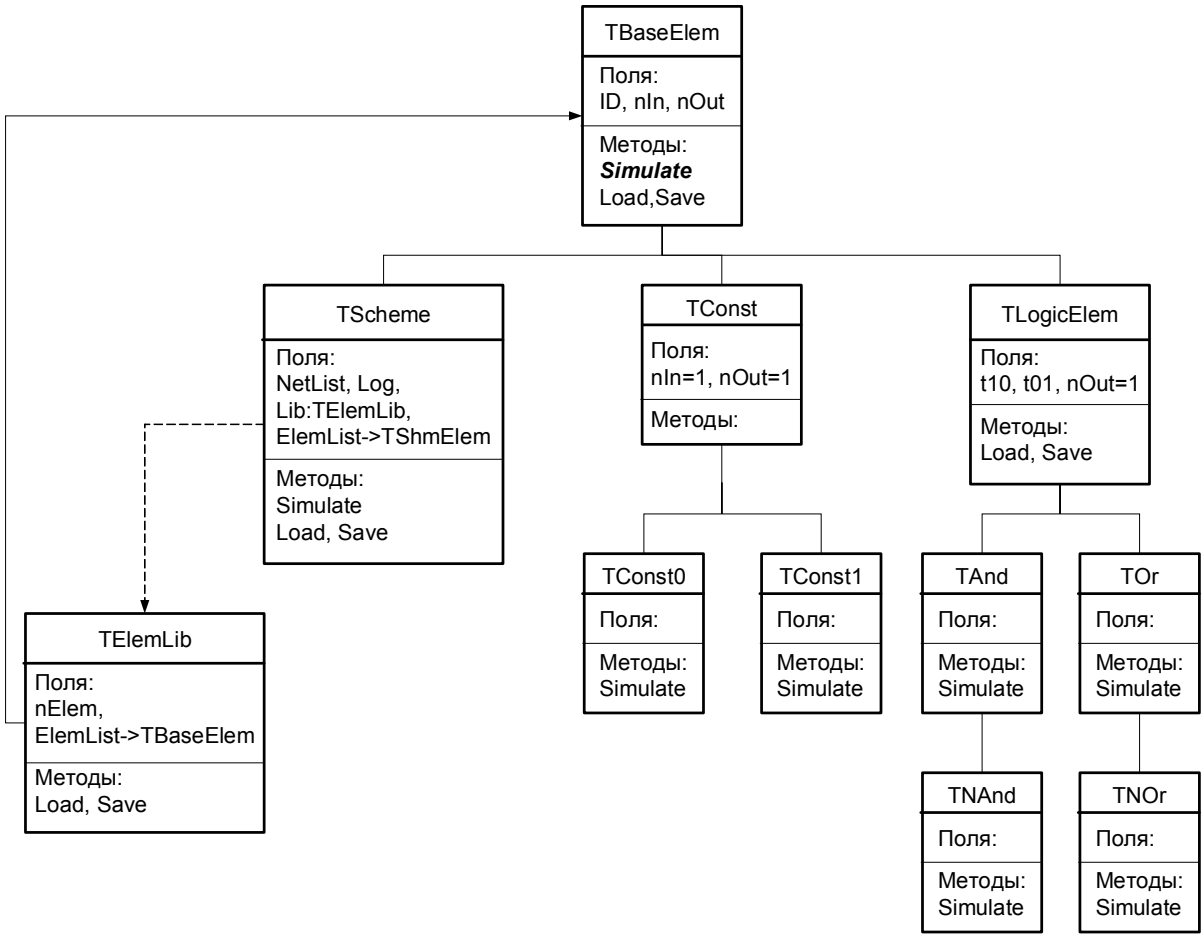

Рисунок 3 – Иерархия классов моделей элементов

Класс TConst соотвествует модели абстрактной константной неисправности (тождественному значению сигнала), а TConst0 и TConst1 моделям элементов с сигналами ≡0 и ≡1 на выходе.

Класс TScheme реализует модуль схемы. Поле NetList определяет список цепей, поле ElemList список элементов (ссылок на объекты из библиотеки элементов схемы), поле Lib – библиотека элементов схемы, поле Log – служебная информация моделирования, используется при построении временных диаграмм (может представлять из себя, например, массив длиной, равной количеству цепей в схеме, элементы которого являются списками записей о появлении в соответствующей цепи определенного значения сигнала в определенный момент времени). Реализация метода Simulate определяется выбором алгоритма моделирования (например, синхронное или асинхронное, событийное или несобытийное).

Класс TElemLib соответствует библиотеке элементов схемы. Поле nElem определяет количество элементов в библиотеке, поле ElemList – список элементов (производных от TBaseElem).

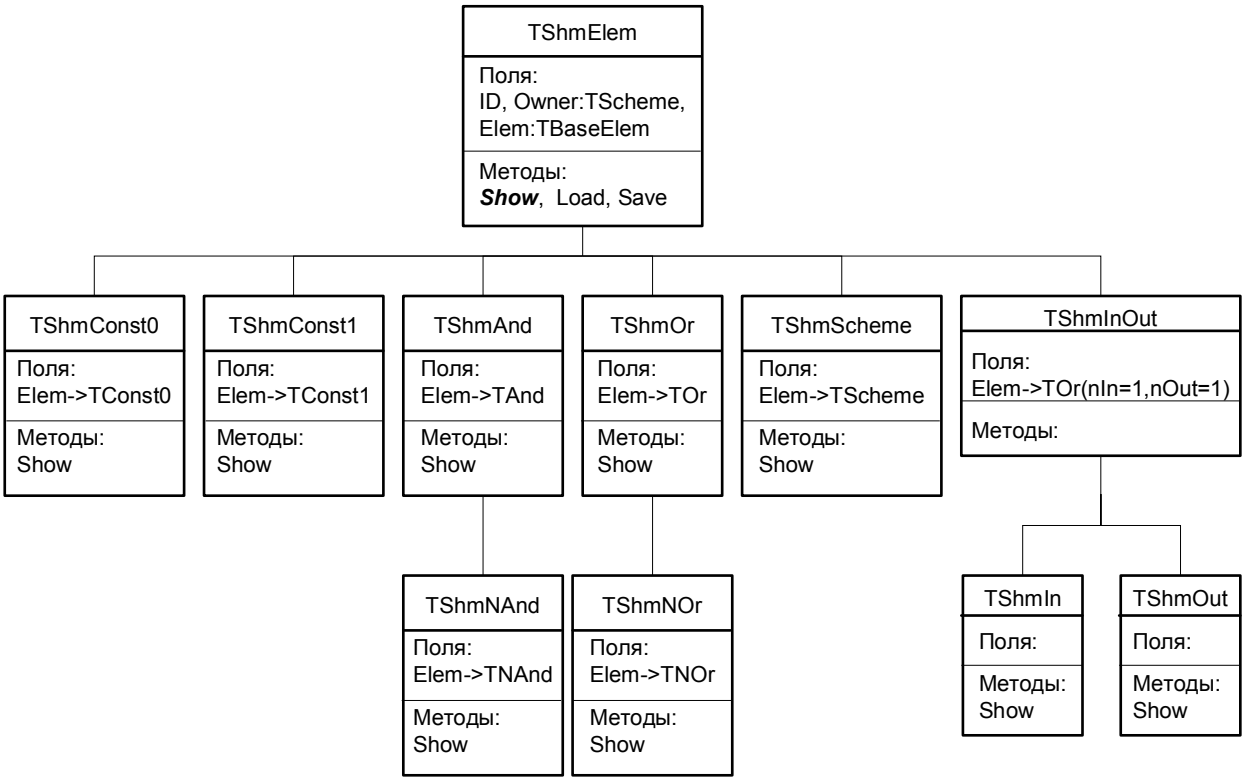

Рисунок 4 – Иерархия классов элементов схемы

На рисунке 4 представлена иерархия классов элементов схемы.

Класс TShmElem является базовым классом и соответствует абстрактному элементу схемы. Основное назначение иерархии – обеспечить механизм хранения информации о типе элемента и отображение элементов на схеме в интерфейсном окне оператора в соответствии с типом элемента. Поле ID определяет функциональное обозначение элемента на схеме, поле Owner – ссылка на схему, к которой относится элемент, поле Elem – ссылка на элемент библиотеки элементов схемы, который и определяет тип. Метод Show предназначен для отображения элемента на схеме, он объявлен абстрактным, но это не обязательно: некоторые действия по отображению элемента вполне могут быть выполнены и на уровне базового класса.

Классы TShmConst0, TShmConst1, TShmAnd, TShmOr, TShmNAnd, TShmNOr, TShmScheme реализуют следующие элементы схемы (в порядке перечисления): ≡0, ≡1, И, ИЛИ, И-НЕ, ИЛИ-НЕ, подсхему.

Класс TShmInOut соответствует фиктивному элементу схемы – внешнему выводу схемы (вывод разъема). Его поле Elem ссылается на повторитель (одновходовой элемент ИЛИ). Этот выбор продиктован не общими соображениями, а, скорее, связан с особенностями выбранного автором алгоритма моделирования, структур данных и т.п. Возможны и другие решения, например, не создавать отдельных элементов для каждого вывода разъема, а один общий для всего разъема, с соответствующим количеством выводов, такое решение может предполагать создание отдельного класса в иерархии моделей элементов. Классы TShmIn и TShmOut соответствуют входному и выходному выводу схемы.

Принцип функциональности

Функции системы в целом рассмотрены в связи с принципом конечной цели. Рассмотрим функции, входные и выходные данные выделенных подсистем.

Основной функцией подсистемы создания моделей логических элементов является, очевидно, создание моделей логических элементов (ЛЭ) на основе данных, вводимых оператором, – описания ЛЭ. Описание ЛЭ есть входные данные для подсистемы, на выходе системы – модель ЛЭ. Поскольку подсистема должна позволять редактирование ранее созданных моделей, то подсистема может также может принимать от библиотеки элементов схемы ранее созданные модели ЛЭ.

Подсистема создания моделей схем должна позволять оператору создавать и редактировать модели схем на основании описания схемы и действий оператора по ее разметке. Выходные данные подсистемы – модель схемы. Элементы схемы также могут быть схемами, то для таких элементов возможно редактирование модели подсхемы, переданной из библиотеки элементов схемы.

Подсистема задания входных наборов и выдачи результатов тестирования дает возможность оператору задать необходимое число входных наборов, инициировать моделирование заданной схемы на этих наборах и проанализировать его результаты на основе конечных значений сигналов на выходных вентилях элементов схемы и времени их установления, а также временных диаграмм работы схемы на заданном входном наборе. Режим отображения результатов моделирования является управляющим параметром подсистемы. Входные данные системы – описание входных наборов, выходные данные подсистемы моделирования, выходные данные подсистемы – входные наборы, передаваемые подсистеме моделирования, конечные значения сигналов на выходных вентилях элементов схемы, время их установления, а также временные диаграммы.

Библиотека элементов схемы определяет элементный базис построения схемы и организует доступ к моделям элементов. На вход библиотеки поступает идентификатор элемента (или подсхемы), а на ее выходе – модель элемента (или подсхемы).

Подсистема моделирования выполняет трехзначное моделирование с задержками. Входными данными для нее являются модель схемы, модели элементов схемы и входной набор, на котором будет производиться моделирование. Выходные данные – это информация о состоянии цепей схемы на протяжении всего интервала моделирования.

Все подсистемы, за исключением подсистемы моделирования, также обмениваются данными с файловой системой при загрузке и сохранении соответствующих данных.

Принцип развития

Проектируемая система может быть расширена следующим образом:

– расширение элементной базы (например, добавление моделей более сложных комбинационных устройств включением нового класса в иерархию моделей элементов);
– расширение предметной области, т.е. моделирование схем с обратными связями (за счет усложнения алгоритма моделирования);
– введение возможности работы с моделями и результатами моделирования не только на уровне файлов, но и с использованием централизованной базы данных.

Принцип сочетания централизации и децентрализации

В множестве выделенных подсистем можно выделить несколько подмножеств (возможно пересекающихся), которые будут обладать достаточно высокой степенью автономности от других подмножеств. Например, можно выполнить декомпозицию таким образом:

– {подсистема создания моделей логических элементов};
– {подсистема создания моделей схем, библиотека элементов схемы};
– {подсистема создания задания входных наборов и выдачи результатов моделирования, подсистема моделирования, подсистема представления модели схемы (как часть подсистемы создания моделей схем), библиотека элементов схемы}.

Такое разбиение позволит реализовать полученные подмножества в виде отдельных исполняемых модулей и физически разделить процессы создания моделей ЛЭ, схем и моделирования в проектируемой системе. В этом случае обмен данными между подсистемами разных подмножеств будет производиться через файловую систему.

С другой стороны, все подсистемы можно реализовать в одном исполняемом модуле, регламентируя порядок обращения к каждой из подсистем посредством интерфейса оператора.

Принцип учета неопределенности и случайностей

В проектируемой системе следует предусмотреть возможность реакции на некорректные с точки зрения системы действия оператора, например:

– попытка повторного использования функционального обозначения элемента;
– несовместимость имеющейся и подгружаемой из файла модели ЛЭ или подсхемы по каким-либо параметрам (к примеру, числу входов);
– попытка создания обратных связей и т.д.

3 БИБЛИОГРАФИЯ

1. Спицнадель В. Н. Основы системного анализа. – СПб.: «Издательский дом «Бизнес-пресса», 2000

2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. – СПб.: Питер, 2002

3. ISO 8402:1994 Quality management and quality assurance - Vocabulary

4. 1061-1998 IEEE Standard for Software Quality Metrics Methodology

5. Workman D. A, Crutchfield R. Quality Guidelines = Designer Metrics Proceedings of the 1994 conference on TRI-Ada '94 November 1994

6. Houdek F., Kempter H. Quality patterns — An approach to packaging software engineering experience ACM SIGSOFT Software Engineering Notes , Proceedings of the 1997 symposium on Symposium on software reusability May 1997.

7. Метрики качества программного обеспечения. - http://www.pmprofy.ru/content/rus/67/672-article.asp