электронное научно-техническое издание
НАУКА и ОБРАЗОВАНИЕ
Эл № ФС 77 - 30569. Государственная регистрация №0421100025. ISSN 1994-0408

Системный анализ и реинжиниринг унаследованного программного обеспечения

 # 04, апрель 2011

авторы: Массель Л. В., Подкаменный Д. В.

УДК 519.685

Иркутск, Институт систем энергетики им. Л.А. Мелентьева (ИСЭМ) СО РАН
massel@isem.sei.irk.ru
pdmitriy@iztm.com

Введение. Проблемы анализа и реинжиниринга унаследованного программного обеспечения исследовались авторами на примере программных комплексов для исследования и поддержки принятия решений в энергетике. Программное обеспечение этой отрасли, созданное в нашей стране преимущественно в период бурного развития отрасли энергетики и появления массовых и промышленных компьютеров на рубеже 1980-1990-х гг., не может эффективно использоваться на современных компьютерах и в современных операционных средах, хотя и представляет большую интеллектуальную ценность. Такая ситуация характерна не только для энергетики, поэтому авторы считают, что предлагаемый ими подход может быть полезен и для других предметных областей, где возникают подобные проблемы.

Многие программные системы создавались с расчетом на системное окружение, которое с течением времени изменилось и эти программные системы устарели. Другие программные системы в процессе поддержки подверглись таким изменениям, что дальнейшее их развитие стало невозможным из-за крайне сложной и запутанной архитектуры. Третьи программные системы изначально создавались без учета тенденций развития информационных технологий и к моменту своего появления уже содержали устаревшие решения.

Подобные программные системы называют «унаследованными программными системами» или просто «унаследованными системами». Существуют подходы к реинжинирингу таких систем, но, поскольку каждый программный комплекс по-своему уникален, найти автоматизированное решение задачи не удается. Важным этапом в реинжиниринге унаследованного программного обеспечения является анализ, на основе которого принимаются решения о стратегии реинжиниринга. В статье рассматривается

предлагаемый авторами метод анализа, который поддержан авторской экспертной системой Expasy.

Унаследованное программное обеспечение. Унаследованные системы (legacy systems) – это системы, по тем или иным причинам переставшие удовлетворять изменившимся потребностям применений, которые, тем не менее, продолжают использоваться ввиду больших затруднений, возникающих при попытке их замены [1].

В [2, 3] дано описание унаследованных систем через присущие им свойства. Унаследованными системами могут быть:

- компьютерные системы, которые по тем или иным причинам перестали устраивать пользователей;
- системы для мэйнфреймов, написанные на Cobol и прослужившие более 5 лет;
- исчерпавшие себя системы для мини-компьютеров или мэйнфреймов;
- приложения с интерфейсом для DOS (Unix);
- совокупность аппаратного и программного обеспечения, которое успешно выполняло возложенные на него задачи до тех пор, пока не пришла пора заменить его новыми средствами;
- любые морально устаревшие системы;
- все вышеперечисленное в совокупности.

Хотя впервые эти свойства унаследованных систем были описаны в 1998 году, уже сейчас можно указать дополнительные признаки, которые характерны для унаследованных систем нашего времени. Унаследованные системы используют морально устаревшие технологии, архитектуры, платформы, а также собственно программное и информационное обеспечение. При проектировании таких систем, как правило, не предусматриваются должные меры для их пошаговой миграции в новые системы. Для таких систем характерны монолитность и закрытость. До последнего времени практически любая система после создания противодействовала изменениям и имела тенденцию быстрого превращения в бремя организации, потому что при ее разработке использовались «устаревшие» технологии, архитектуры, платформы.

Считается, что унаследованная система потенциально проблематична по следующим причинам:

- унаследованная система обычно функционирует на устаревшем оборудовании, и поддержка технической части такого оборудования с течением времени будет все больше и больше затрудняться;

- унаследованные системы трудно поддерживать – улучшение и расширение системы, а также исправление ошибок зачастую невозможно из-за недопонимания разработчиками внутреннего устройства системы и отсутствия по многим причинам (например, уход из организации) непосредственных разработчиков этой системы. Такое недопонимание может быть вызвано недостаточной документированностью унаследованной системы или полной потерей документации к системе;
- интеграция унаследованной системы с новыми системами также может быть затруднена либо невозможна из-за различий в используемых базовых технологиях.

Анализ унаследованного программного обеспечения. Для применения метода системного анализа унаследованной программной системы в первую очередь необходимо убедиться, что унаследованное программное обеспечение обладает свойствами системы, а также определить метод декомпозиции и критерии оценки характеристик унаследованной системы для последующего анализа.

Работы по анализу программных систем начали появляться с момента усложнения технологии программирования и перехода к использованию языков высокого уровня. Классические методы анализа программных систем описаны в [4, 5], дальнейшее продолжение предложенные методы получили в работах [6-8]. Отечественная наука достаточно давно обратилась к системному анализу программного обеспечения. Первые работы по анализу программ появились, благодаря разработке сложных пакетов программ [9, 10]. Глубокий системный анализ программного обеспечения дается также в работе [11].

В то же время, несмотря на то, что анализу программного обеспечения посвящено много работ, практически отсутствуют методы системного анализа унаследованного программного обеспечения с целью последующего реинжиниринга. Основная направленность существующих методов заключается в способах разработки программного обеспечения, а также в методах оценки трудоемкости этих разработок, в то время как для решения проблемы унаследованного программного обеспечения в первую очередь важен анализ уже существующих программ.

Опираясь на приведенные выше работы, авторы предлагают метод анализа унаследованного программного обеспечения, основанный на декомпозиции программ на программные модули, количественном анализе модулей и структурном анализе полученной модели.

Декомпозиция унаследованных систем может выполняться на различных уровнях абстракции. Существует общепринятый вид абстракции программных систем, в разных источниках обозначаемый как структура или архитектура программного обеспечения

(системы). *Архитектура программного обеспечения* – это первичная организация системы, сформированная ее компонентами, отношениями между компонентами и внешней средой системы, а также принципами, определяющими дизайн и эволюцию системы [12]. *Структура* программы – определение всех модулей программы, их иерархий и сопряжений между ними [13]. Уровень абстракции определяется степенью детализации описания модулей программы.

Программный модуль – это выделенная по тем или иным мотивам часть первичного материала программного фонда [11], а также любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса [14]. Иными словами, модуль – это замкнутая программа, которую можно вызвать из любого другого модуля программы и можно отдельно компилировать. Таким образом, можно называть модулем унаследованной системы на высшем уровне абстракции программу из пакета, составляющего унаследованную систему или отдельную процедуру или функцию (в терминах языка С) на низшем уровне абстракции.

Целесообразно выделить три уровня абстракции структуры программных комплексов: модули – исполняемые файлы системы, динамические библиотеки, СУБД, файлы данных; модули – объектные файлы системы; модули – функции, память. Вне зависимости от уровня абстракции в описании структуры унаследованной системы модуль обладает тремя основными атрибутами: 1) выполняет одну или несколько функций; 2) обладает некоторой логикой; 3) используется в одном или нескольких контекстах.

Функция модуля – это внешнее описание поведения модуля. Функция описывает, что делает модуль при обращении, но не раскрывает внутреннее устройство модуля. Логика модуля – это описание внутреннего поведения модуля при выполнении. Контекст модуля – это описание конкретного применения модуля. Модуль может выполнять несколько функций, иметь различную логику поведения, а также использоваться в нескольких контекстах в рамках унаследованной системы.

Определим критерии оценки модулей. Впервые критерии для оценки модуля были предложены Хольтом [6]: хороший модуль снаружи проще, чем внутри; хороший модуль проще использовать, чем построить. Однако очевидно, что эти критерии не удовлетворяют принципу количественной оценки модуля и пригодны лишь как общие рекомендации при создании модульной программы. С позиции системного анализа унаследованных систем больше подходят оценки, предложенные Г. Майерсом в [13], так для оценки программного модуля по Майерсу используются следующие характеристики: размер модуля; прочность

модуля; сцепление с другими модулями; рутинность модуля (независимость от предыстории обращений к нему); объем доступа к данным.

Размер модуля измеряется числом содержащихся в нем операторов или строк кода. *Прочность (связность) модуля* – это мера его внутренних связей [15-17]. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля или силы связности (СС) применяется упорядоченный набор из семи классов.

1. Модуль, прочный по совпадению (СС=0) – самая слабая степень прочности. Между элементами такого модуля нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется как отдельный модуль. Необходимость изменения этой последовательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным.
2. Модуль, прочный по логике (СС=1) – модуль, содержащий набор функций, конкретный выбор выполняемой функции осуществляется вызывающим модулем.
3. Модуль, прочный по классу, а также временная связность (СС=3) – модуль, выполняющий некоторые функции, отнесенные разработчиком к одному классу или части модуля, необходимые в один и тот же период работы системы.
4. Процедурно прочный модуль (СС=5) – модуль, который выполняет несколько функций, отнесенных к одной функциональной процедуре решения задач, при этом части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.
5. Коммуникационно прочный модуль (СС=7) – процедурно прочный модуль, все функции которого связаны по данным.
6. Функционально прочный модуль (СС=10) – модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули.
7. Информационно прочный модуль (СС=9) – модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных

модулей с высшей степенью прочности. Информационно прочный модуль может реализовывать, например, абстрактный тип данных.

Можно предложить, с учетом [18], следующий **алгоритм определения степени прочности модуля**.

1. Если модуль единичная проблемно-ориентированная функция, то это – функционально прочный модуль (СС=10); конец алгоритма. В противном случае перейти к п. 2.
2. Если действия внутри модуля связаны, то перейти к п. 3. Если действия внутри модуля никак не связаны, то перейти к п. 6.
3. Если действия внутри модуля связаны данными, то перейти к п. 4. Если действия внутри модуля связаны потоком управления, перейти к п. 5.
4. Если порядок действий внутри модуля важен, то степень прочности – информационная (СС=9). В противном случае степень прочности – коммуникационная (СС=7). Конец алгоритма.
5. Если порядок действий внутри модуля важен, то степень прочности – процедурная (СС=5). В противном случае степень прочности по классу (СС=3). Конец алгоритма.
6. Если действия внутри модуля принадлежат к одной категории, то степень прочности по логике (СС=1). Если действия внутри модуля не принадлежат к одной категории, то степень прочности – по совпадению (СС=0). Конец алгоритма.

Возможны случаи, когда с модулем ассоциируются несколько степеней прочности. В этих случаях обычно модулю присваивают наибольшую степень прочности.

Сцепление модуля – это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления (СЦ) применяется упорядоченный набор из шести видов сцепления модулей [15].

1. Сцепление по содержимому (СЦ=9). Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле).
2. Сцепление по общей области (СЦ=7). Сцепление модулей, при котором несколько модулей используют одну и ту же область памяти. Такой вид сцепления модулей реализуется, например, при программировании на языке ФОРТРАН с использованием блоков COMMON.
3. Сцепление по внешним данным (СЦ=5). Сцепление модулей по ссылке на глобальный элемент данных.

4. Сцепление по управлению (СЦ=4). Один из модулей явно управляет функционированием другого. Сцепление по управлению обычно сопутствует прочностью по логике этих модулей.
5. Сцепление по формату (СЦ=3). Модули сцеплены по формату, если они ссылаются на одну и ту же неглобальную структуру данных.
6. Сцепление по данным (СЦ=1). Модули сцеплены по данным, когда модуль при вызове другого модуля передает и принимает набор элементарных данных.

Модули могут быть сцеплены несколькими видами сцепления, в таком случае для оценки выбирается сильнейшее из всех. Высокая прочность и слабое сцепление способствуют независимости модулей, поскольку они сводят к минимуму их взаимодействие.

Рутинность модуля – это его независимость от предыстории обращений к нему. Модуль называется рутинным, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль называется зависящим от предыстории, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему.

Размер модуля – LOC (Lines of Code) – означает количество строк исходного кода модуля, эта метрика отражает трудозатраты, которые потребовались для создания этого модуля и, соответственно, трудозатраты, необходимые для понимания и переработки модуля.

Несмотря на то, что указанные характеристики достаточно полно описывают программные модули, для анализа программного обеспечения научных исследований их может оказаться недостаточно, поэтому предлагается добавить еще одну метрику, предложенную МакКейбом – *цикломатическую сложность модуля*, которая равна цикломатическому числу потокового графа модуля, уменьшенному на единицу, другими словами – цикломатическая сложность равна количеству условных операторов и операторов ветвления в модуле.

Между размером модуля и сложностью можно установить связь. Так, Хольстед в [19] вводит новую метрику – объем модуля $V = N\log_2 n$, N – общее число всех идентификаторов, а n – число различных идентификаторов. Однако полученная метрика не в полной мере отражает размер и сложность модуля, поэтому применять её следует с осторожностью.

Таким образом, модель модуля может быть представлена следующей кортежной записью:

$$M = \{R, S, P, A, T, C, I, O, UI\},$$

где R – размер модуля, LOC-оценка, S – сложность модуля по МакКэйбу, P – прочность модуля, сила связности по Майерсу, $A \in B$ – рутинность модуля. Члены *T, C, I, O, UI* из B – описывают поведение модуля – логику: T – трансформирующая логика, C – вычислительная, I и O – процедуры чтения/записи из/в среду, UI – ожидающий ввод от пользователя.

Операция упрощения структурной модели программной системы – композиция модулей:

$$M_1 \otimes M_2 = \{R_1 + R_2, S_1 + S_2, \min\{P_1, P_2\}, A_1 \vee A_2, T_1 \vee T_2, C_1 \vee C_2, I_1 \vee I_2, O_1 \vee O_2, UI_1 \vee UI_2\}$$

Операция детализации структурной модели – декомпозиция модулей – выполняется по алгоритму определения характеристик итоговых модулей.

Для наглядного представления структурной модели предлагается использовать CASE-нотацию. Каждый модуль изображается в виде бокса со следующими полями: название модуля, размер, сложность, прочность, логика. Связи между модулями изображаются направленной стрелкой, означающей зависимость между модулями, стрелка маркируется числом, означающим силу связности между модулями. Рутинность модуля обозначается пунктирной границей бокса.

Методика и инструментальная поддержка реинжиниринга. Опираясь на изложенный выше метод анализа унаследованного программного обеспечения, авторы предлагают методику реинжиниринга, которая в общем виде представлена на рис. 1 [20].

Систематический анализ реинжиниринга проводится над унаследованной системой до тех пор, пока «оценка попытки» (рис. 2) не выдаст такие альтернативы, что на шаге «анализ решения» (рис. 3) не будет выбрана одна из альтернатив: поддержка текущего состояния унаследованной системы или разработка новой системы.

На основе предложенного метода анализа, разработанной методики реинжиниринга и опыта их применения для реинжиниринга ряда программных комплексов совместно с коллегами из лаборатории «Информационные технологии в энергетике», возглавляемой Л.В. Массель, разработана экспертная система поддержки реинжиниринга Expacy.

Экспертная система Expacy на основе полученных от пользователя знаний об унаследованной системе и пожеланий к требуемой системе, применяя правила вывода из собственной базы знаний, выстраивает конкретные стратегии реинжиниринга нализируемого унаследованного программного обеспечения. Экспертная система разработана с использованием языка и оболочки CLIPS. База знаний построена на основе исследования

результатов опроса экспертов и программистов, занимающихся проблемами унаследованного программного обеспечения.

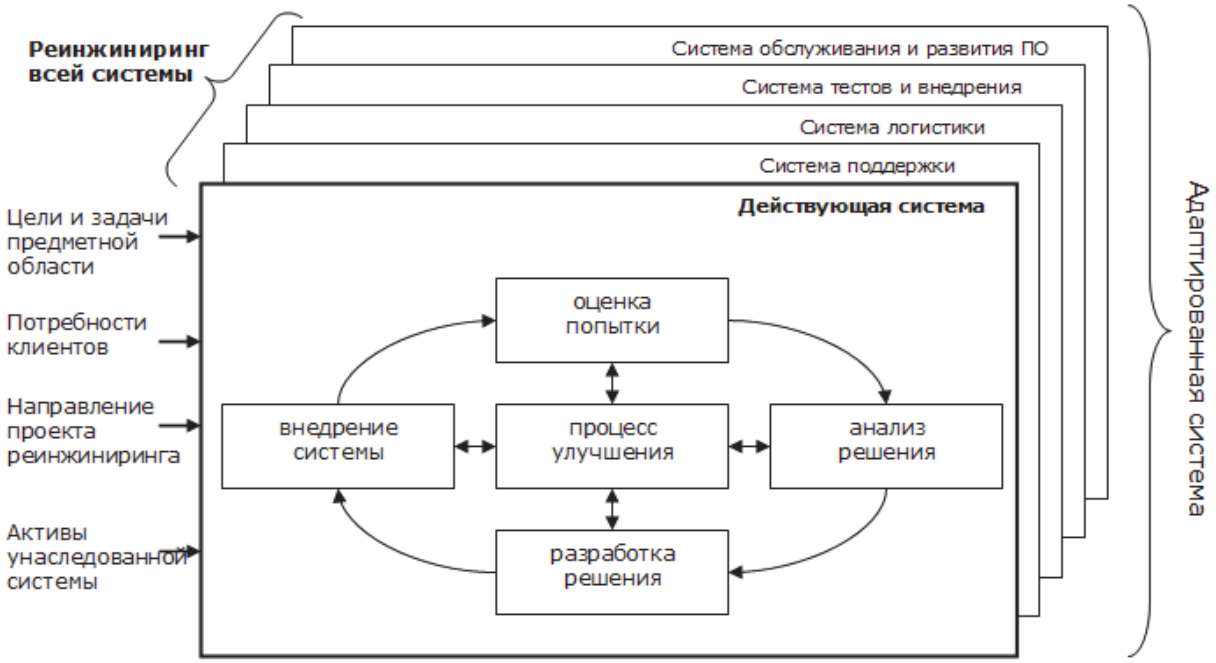

Рис. 1. Структура реинжиниринга

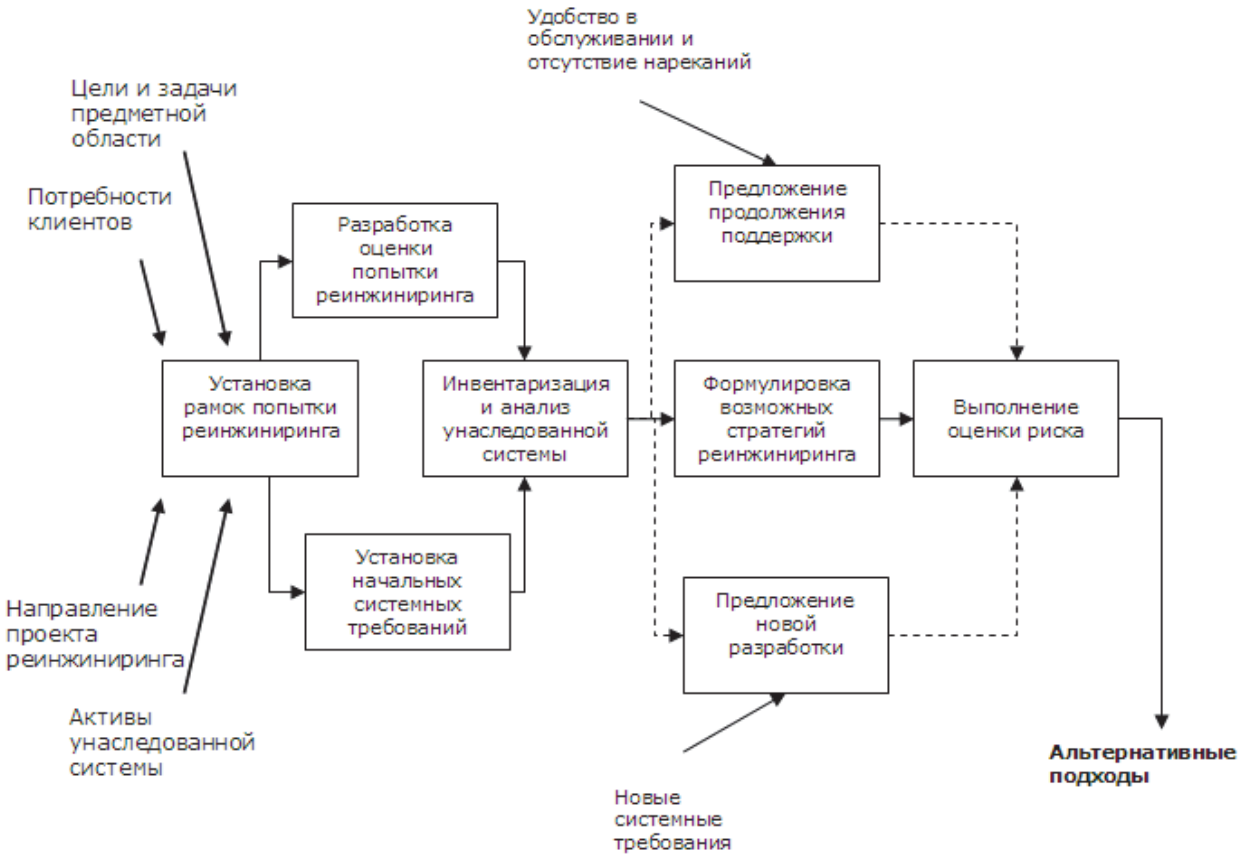

Рис. 2. Оценка попытки реинжиниринга

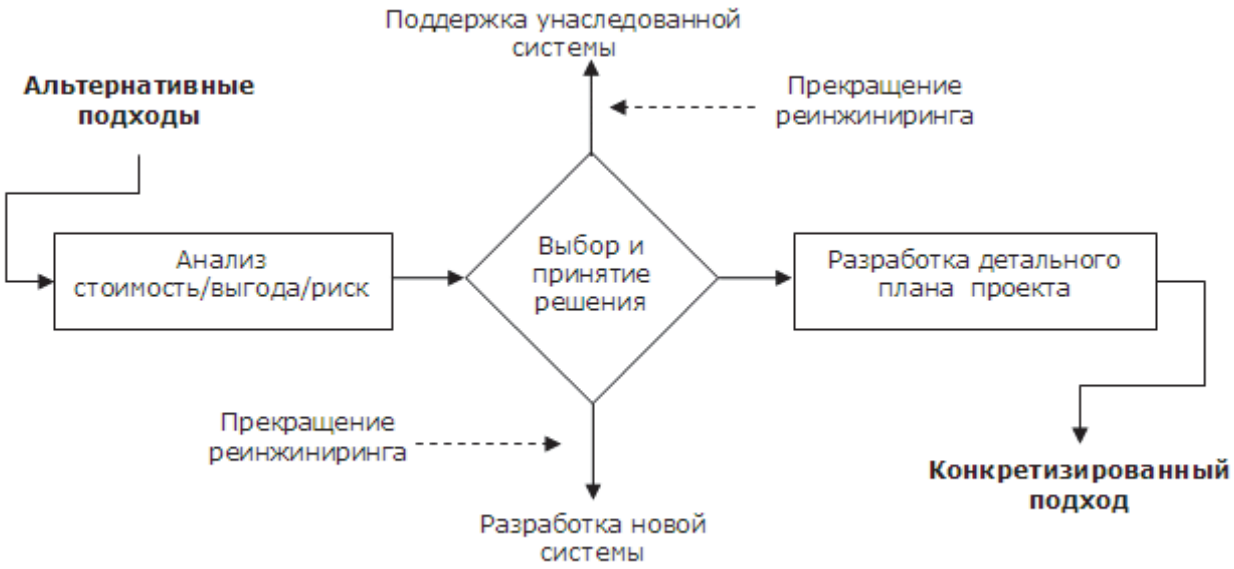

Рис. 3. Анализ решения

Для дальнейшего проведения реинжиниринга может быть использован один из инструментальных инструментов реинжиниринга для работы с исходным кодом унаследованных систем, например, Undestand (www.scitools.com) или Doxigen. Первая предоставляет больше функций по работе с исходным кодом, вторая предпочтительнее из-за своего свободного распространения.

Заключение. Предложенные метод анализа унаследованного программного обеспечения и экспертная система Expacy применялись в ИСЭМ СО РАН для реинжиниринга пакета прикладных программ OPTCON и программного комплекса ИНТЭК для исследований направлений развития топливно-энергетического комплекса страны с учетом требований энергетической безопасности, а также для анализа программного комплекса ГАРМОНИКИ для расчета, анализа и исследования свойств режимов высших гармоник и программно-вычислительного комплекса «Янтарь» для оценки надежности электроэнергетических систем. В ходе анализа были установлены недостатки архитектуры, которые препятствуют реинжинирингу этих программных комплексов. Так, для ПК «Янтарь» были определены модули, обладающие слабой степенью прочности, большой силой сцепления, разнообразной внутренней логикой. ППП OPTCON в результате анализа был подвергнут переработке с целью улучшения структуры программы и в виде обновленного программного пакета послужил ядром вычислительного сервера для решения задачи оптимального управления OPTCON [21].

Результаты, описанные в статье, получены при частичной финансовой поддержке грантов РФФИ №10-07-00264 и №11-07-00192, а также гранта Программы Президиума РАН №2.29.

Литература

1. Брюхов Д.О., Задорожный В.И., Калиниченко Л.А., Курошев М.Ю., Шумилов С.С. Интероперабельные информационные системы: архитектуры и технологии // СУБД – 1995. – №4.
2. Bisbal, J., Lawless, D., Wu, B., Grimson, J. Legacy Information System Migration: A Brief Review of Problems, Solutions and Research Issues. // IEEE Software, 16, 1999. 103-111.
3. Энн Маккрори. Что такое унаследованные системы? // Computerworld – 1998. – №14.
4. E. W. Dijkstra. The structure of "THE"-multiprogramming system. // Comm. ACM, Vol. 11, No. 5, May 1968, pp. 341-346.
5. D. Parnas. On the criteria to be used in decomposing systems into modules. // Communications of the ACM, 15(12):1053-1058.
6. R.C.Holt. Structure of Computer Programs: A Survey// Proceedings of the IEEE, 1975, 63(6). - P. 879-893.
7. G.J. Mayers. Reliable Software Through Composite Design. – Petrocelli/Charter edition, in English - 1st ed., 1975.
8. E. Yourdon, L.L. Constantine. Structured Design. – Yourdon Press, 1975.
9. Е.А. Жоголев. Система программирования с использованием библиотеки подпрограмм. // Система автоматизация программирования. – М. : Физматгиз, 1961.
10. Ф.Я. Дзержинский, А.И. Тер-Сааков. Технология программирования – структурный подход. — М.: Цнииатоминформ, 1978.
11. М.М. Горбунов-Посадов. Конфигурации программ. Рецепты безболезненных изменений. — 2-е изд., испр. и доп. — М.: Малип, 1994.
12. Recommended Practice for Architectural Description of Software-Intensive Systems, ANSI/IEEE Std 1471-2000
13. Майерс Г. Надежность программного обеспечения. М.:Мир, 1980. – 359 с.
14. Зубкова Т.М. Технология разработки программного обеспечения. Оренбург: ГОУ ОГУ, 2004. – 101 с.
15. Page-Jones, M. The Practical Guide to Structured Systems Design. Englewood Cliffs, NY: Yourdon Press, 1988.
16. Stevens, W., Myers, G., and Constantine, L. Structured Design // IBM Systems Journal, Vol. 13(2), 1974, pp. 115-139.
17. Yourdon, E., and Constantine, L. Structured Design: fundamentals of a discipline of computer program and systems design. Englewood Cliffs, NJ: Prentice-Hall, 1979.

18. Орлов С.А. Технологии разработки программного обеспечения. СПб.:Питер, 2002. – 322 с.
19. Halstead, M.H. Software Physics Comparison of a Sample Program in DSL Alpha and COBOL, IBM, Res.R. RJ1460, San Jose CA, 24, 1974.
20. Подкаменный Д.В. Методика адаптации унаследованных систем и ее инструментальная поддержка / Информационные технологии в науке и управлении / Труды XIII Байкальской Всероссийской конференции, ч. 2. – Иркутск: ИСЭМ СО РАН, 2008. – С.211-219.
21. Подкаменный Д.В. Вычислительный сервер для решения задач оптимального управления / Современные технологии. Системный анализ. Моделирование. Иркутск: ИрГУПС. – 2008.- №3 (19) . – С. 109-113.