Лекция 5. Модели жизненного цикла ПО, проектирование как конструирование.

Аннотация.

Стандарты жизненного цикла ПО

Информационные системы должны удовлетворять интересам бизнеса, а также быть легко модифицируемыми и недорогими. Плохо спроектированная система, в конечном счете, требует больших затрат и времени для ее содержания и обновления.

Одним из базовых понятий методологии проектирования ИС является понятие жизненного цикла ее программного обеспечения (ЖЦ ПО).

Жизненный цикл ПО – это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

Существует целый ряд стандартов, регламентирующих ЖЦ ПО, а в некоторых случаях и процессы разработки.

Среди наиболее известных стандартов можно выделить следующие:

- ***ГОСТ 34.601-90*** - распространяется на автоматизированные системы и устанавливает стадии и этапы их создания. Кроме того, в стандарте содержится описание содержания работ на каждом этапе. Стадии и этапы работы, закрепленные в стандарте, в большей степени соответствуют каскадной модели жизненного цикла.

- ***ISO/IEC 12207:1995*** - стандарт на процессы и организацию жизненного цикла. Распространяется на все виды заказного ПО. Стандарт не содержит описания фаз, стадий и этапов.

- ***Custom Development Method*** (методика Oracle) по разработке прикладных информационных систем - технологический материал, детализированный до уровня заготовок проектных документов, рассчитанных на использование в проектах с применением Oracle. Применяется CDM для классической модели ЖЦ (предусмотрены все работы/задачи и этапы), а также для технологий "быстрой разработки" (Fast Track) или "облегченного подхода", рекомендуемых в случае малых проектов.

- ***Rational Unified Process (RUP)*** предлагает итеративную модель разработки, включающую четыре фазы: начало, исследование, построение и внедрение. Каждая фаза может быть разбита на этапы (итерации), в результате которых выпускается версия для внутреннего или внешнего использования. Прохождение через четыре основные фазы называется циклом разработки, каждый цикл завершается генерацией версии системы. Если после этого работа над проектом не прекращается, то полученный продукт продолжает развиваться и снова минует те же фазы. Суть работы в рамках RUP - это создание и сопровождение моделей на базе UML.

- ***Microsoft Solution Framework (MSF)*** сходна с RUP, так же включает четыре фазы: анализ, проектирование, разработка, стабилизация, является итерационной, предполагает использование объектно-ориентированного моделирования. MSF в сравнении с RUP в большей степени ориентирована на разработку бизнес-приложений.

- ***Extreme Programming (XP).*** Экстремальное программирование (самая новая среди рассматриваемых методологий) сформировалось в 1996 году. В основе

методологии командная работа, эффективная коммуникация между заказчиком и исполнителем в течение всего проекта по разработке ИС, а разработка ведется с использованием последовательно дорабатываемых прототипов.

Основные процессы жизненного цикла ISO/IEC 12207

В соответствии с базовым международным стандартом ISO/IEC 12207 все процессы ЖЦ ПО делятся на три группы:

1. **Основные процессы:**
    - приобретение;
    - поставка;
    - разработка;
    - эксплуатация;
    - сопровождение.
2. **Вспомогательные процессы:**
    - документирование;
    - управление конфигурацией;
    - обеспечение качества;
    - разрешение проблем;
    - аудит;
    - аттестация;
    - совместная оценка;
    - верификация.
3. **Организационные процессы:**
    - создание инфраструктуры;
    - управление;
    - обучение;
    - усовершенствование.

В таблице 1 приведены ориентировочные описания основных процессов ЖЦ. Вспомогательные процессы предназначены для поддержки выполнения основных процессов, обеспечения качества проекта, организации верификации, проверки и тестирования ПО. Организационные процессы определяют действия и задачи, выполняемые как заказчиком, так и разработчиком проекта для управления своими процессами.

Для поддержки практического применения стандарта ISO/IEC 12207 разработан ряд технологических документов:

- Руководство для ISO/IEC 12207 (ISO/IEC TR 15271:1998 Information technology - Guide for ISO/IEC 12207)
- Руководство по применению ISO/IEC 12207 к управлению проектами (ISO/IEC TR 16326:1999 Software engineering - Guide for the application of ISO/IEC 12207 to project management).

Таблица 1. Содержание основных процессов ЖЦ ПО ИС (ISO/IEC 12207)

Процесс (исполнитель процесса)	**Действия**	**Вход**	**Результат**
Приобретение (заказчик)	• Инициирование • Подготовка заявочных предложений • Подготовка договора • Контроль деятельности поставщика • Приемка ИС	• Решение о начале работ по внедрению ИС • Результаты обследования деятельности заказчика • Результаты анализа рынка ИС/ тендера • План поставки/ разработки • Комплексный тест ИС	• Технико-экономическое обоснование внедрения ИС • Техническое задание на ИС • Договор на поставку/ разработку • Акты приемки этапов работы • Акт приемно-сдаточных испытаний
Поставка (разработчик ИС)	• Инициирование • Ответ на заявочные предложения • Подготовка договора • Планирование исполнения • Поставка ИС	• Техническое задание на ИС • Решение руководства об участии в разработке • Результаты тендера • Техническое задание на ИС • План управления проектом • Разработанная ИС и документация	• Решение об участии в разработке • Коммерческие предложения/ конкурсная заявка • Договор на поставку/ разработку • План управления проектом • Реализация/ корректировка • Акт приемно-сдаточных испытаний

Разработка (разработчик ИС)	• Подготовка • Анализ требований к ИС • Проектирование архитектуры ИС • Разработка требований к ПО • Проектирование архитектуры ПО • Детальное проектирование ПО • Кодирование и тестирование ПО • Интеграция ПО и квалификационное тестирование ПО • Интеграция ИС и квалификационное тестирование ИС	• Техническое задание на ИС • Техническое задание на ИС, модель ЖЦ • Подсистемы ИС • Спецификации требования к компонентам ПО • Архитектура ПО • Материалы детального проектирования ПО • План интеграции ПО, тесты • Архитектура ИС, ПО, документация на ИС, тесты	• Используемая модель ЖЦ, стандарты разработки • План работ • Состав подсистем, компоненты оборудования • Спецификации требования к компонентам ПО • Состав компонентов ПО, интерфейсы с БД, план интеграции ПО • Проект БД, спецификации интерфейсов между компонентами ПО, требования к тестам • Тексты модулей ПО, акты автономного тестирования • Оценка соответствия комплекса ПО требованиям ТЗ • Оценка соответствия ПО, БД, технического комплекса и комплекта документации требованиям ТЗ

Погонин – Интегрированные системы проектирования ИС (35-37).

ISO/IEC 12207 (ISO – International Organization of Standardization – Международная организация по стандартизации; IEC – International Electrotechnical Commission –

Международная комиссия по электротехнике) определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО.

Верификация – это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки исходным требованиям. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникают проблемы учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в проекте стандарта ***ISO 12207-2***.

Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. ЖЦ ПО носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

Модели жизненного цикла ПО

Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО (под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и условий, в которых последняя создается и функционирует). Его регламенты являются общими для любых моделей ЖЦ, методологий и технологий разработки. Стандарт ISO/IEC 12207 описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

В изначально существовавших однородных ИС каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет

полностью завершена работа на текущем (рис. 2.1). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Рис. 2.1 Каскадная схема разработки ПО

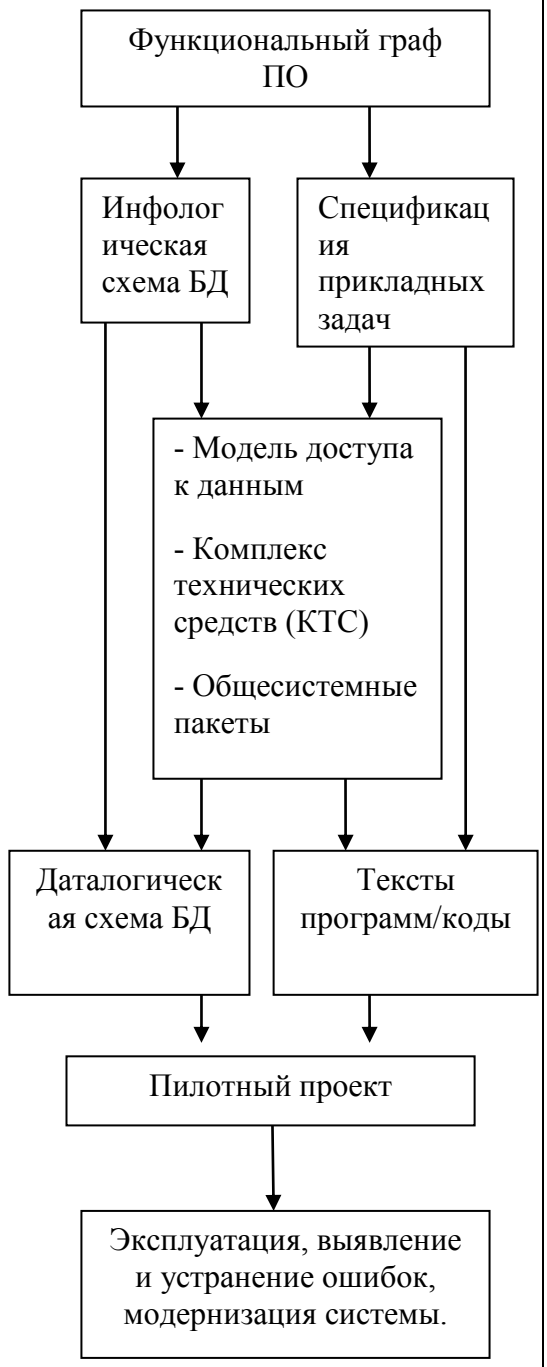
Положительные стороны применения каскадного подхода заключаются в следующем:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении ИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем, чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал вид, представленный на рис. 2.2.

Рис. 2.2 Реальный процесс разработки ПО по каскадной схеме

Рассмотрим более подробно каскадную модель:

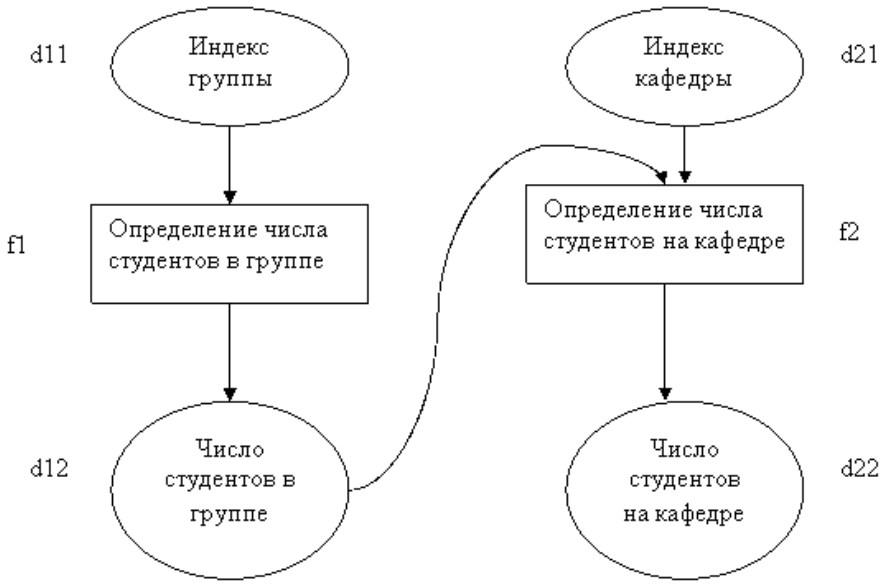
Этап	Результат
1. Выявление информационных потребностей конечных пользователей (предпроектное обследование, разработка ТЗ, частных ТЗ) 2. Концептуальный проект (эскизное проектирование) 3. Архитектура системы (технический проект) 4. Логическое проектирование (техническое проектирование) 5. Комплексная отладка (разработка рабочей документации) 6. Сопровождение.	

Особенность модели: каждый следующий этап проектирования начинается после полного завершения работ по предыдущему этапу.

1. **Выявление информационных потребностей конечных пользователей**

Функциональный граф ПО - граф, узлы которого обозначают данные и процессы будущей системы. Дуги используются для обозначения входных/выходных данных для процесса.

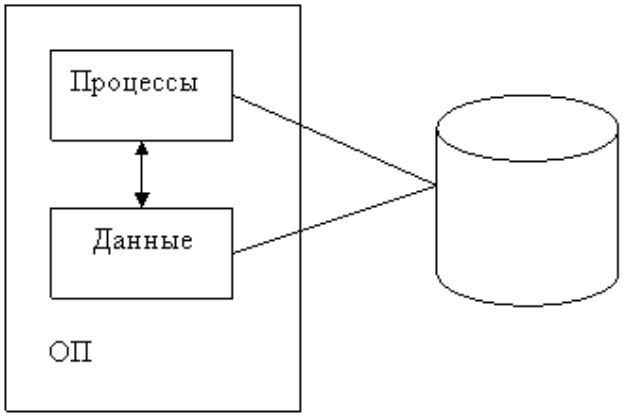
Пример:



d22=f2(d12,d21)=f2(f1(d11),d21)

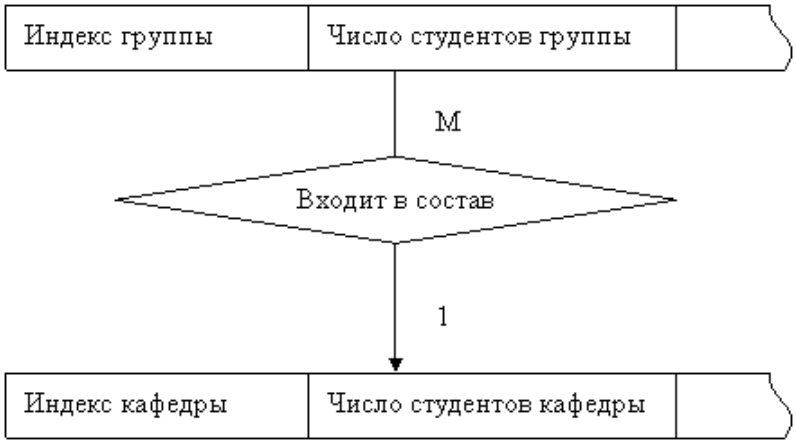
В функциональном графе данные и процессы объединены и в принципе его достаточно для реализации будущей системы (см. формулу к рисунку), но современные компьютеры есть машины Фон-Неймана, где предполагается разделение процессов и данных.

2. Концептуальный проект



Для реализации концептуальной модели проектировщик вынужден выделять из функционального графа данные и строить для них схему БД, а также выделять процессы и разрабатывать для них спецификацию и кодировать. Это является источником большинства ошибок проектирования. Данные функционального графа структурируются в виде инфологической схемы БД.

Пример: (Инфологическая модель БД в нотации Чена)



Спецификация процессов - входные и выходные данные процессов, а также алгоритмическая связь между ними. Для описания спецификации существуют различные методы: структурированный естественный язык (часто используется), язык проектирования спецификации Flow-Form (визуальные языки).

Концептуальный проект не зависит от архитектуры!

3. Выбор архитектуры

- выбор модели доступа к данным (файл-сервер, сервер-БД, сервер-приложение, доступ к данным по Internet/Intarnet)

- выбор комплекса технических средств (выбор «железа»)

- выбор общесистемных пакетов

-выбор способа тиражирования данных

4. Логическое проектирование

Выполняется отражение концептуального проекта в СУБД-ориентированную среду с помощью выбранных оболочек программирования. Сущности преобразуются в таблицы, а на основе спецификации задач разрабатываются тексты программ

Логический проект зависит от архитектуры (можно считать временные характеристики)

5. Отладка

Результаты проектирования БД и приложений объединяются. В итоге разрабатывается пилотный проект системы

6. Сопровождение

Выявление ошибок и их устранение, модернизация.

Достоинства каскадной модели:

- проста, естественна, имеет некоторую привязку к ГОСТу

Недостатки:

- достаточно продолжительный цикл разработки по времени (система морально устаревает)

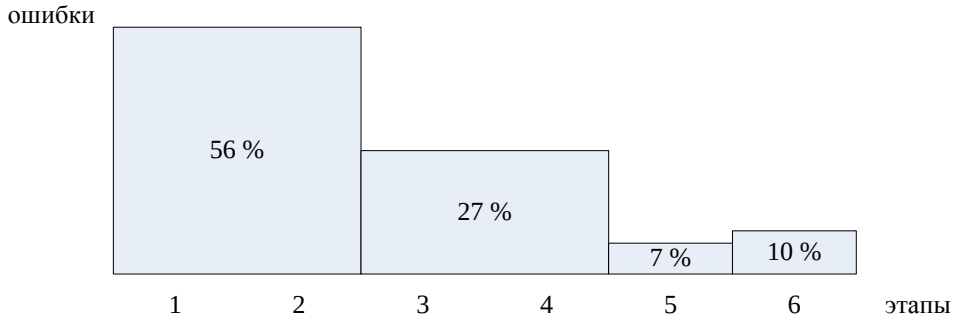
- доработка системы связана с большим объемом перепрограммирования (из-за слабого использования CASE-средств)

Основным недостатком каскадного подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к ИС «заморожены» в виде технического задания на все время ее создания.

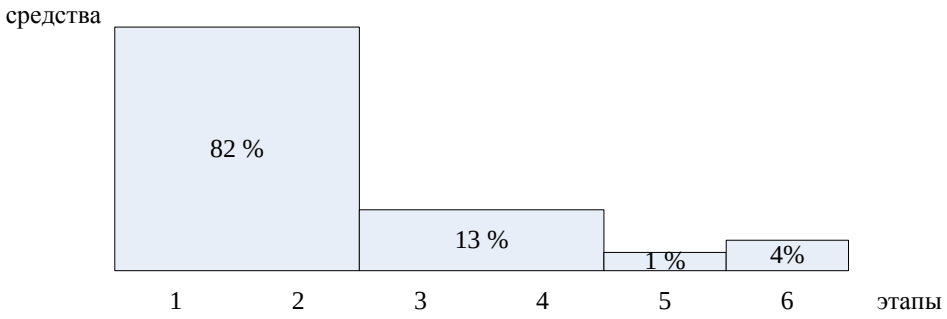
Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

Результаты исследований Д.Мартина (сер. 80х)

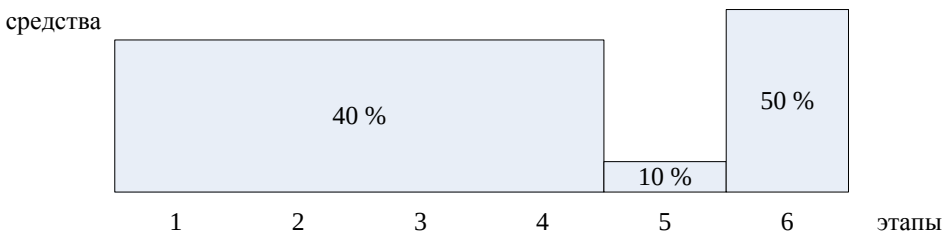
1-я диаграмма: распределение ошибок и просчетов по этапам проектирования, выявленных при сопровождении системы



2-я диаграмма: распределение затрат на исправление ошибок и просчетов, выявленных при сопровождении



3-я диаграмма: распределение трудозатрат по этапам проектирования



Почти половина трудозатрат приходится на устранение ошибок, допущенных на первых 2-х этапах.

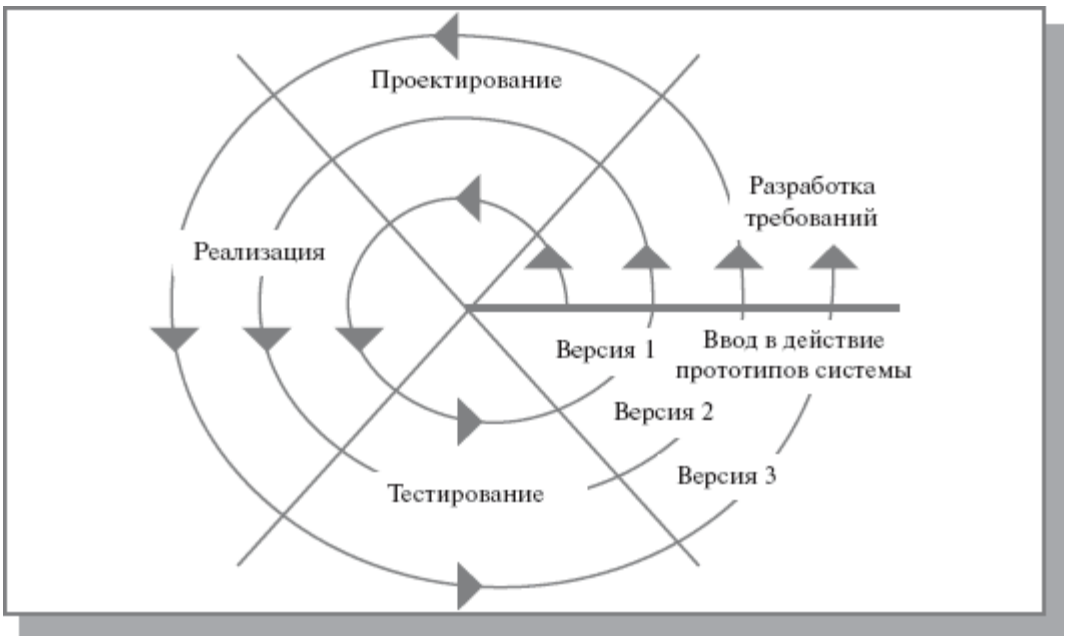
На основании исследований Мартин сформулировал законы:

1. **закон неопределенности в информатике:** процесс автоматизации задачи меняет представление пользователя об этой задаче, т.е. пользователь решает задачу с использованием средств автоматизации иначе, чем без них (пользователя надо использовать постоянно в процессе проектирования, а не только в начале)
2. чем больше времени прошло с момента совершения ошибки до момента ее обнаружения, тем больше средств необходимо для ее устранения (смотри диаграмму 2)
3. программисты и проектировщики не учатся на чужих ошибках, а только на своих.

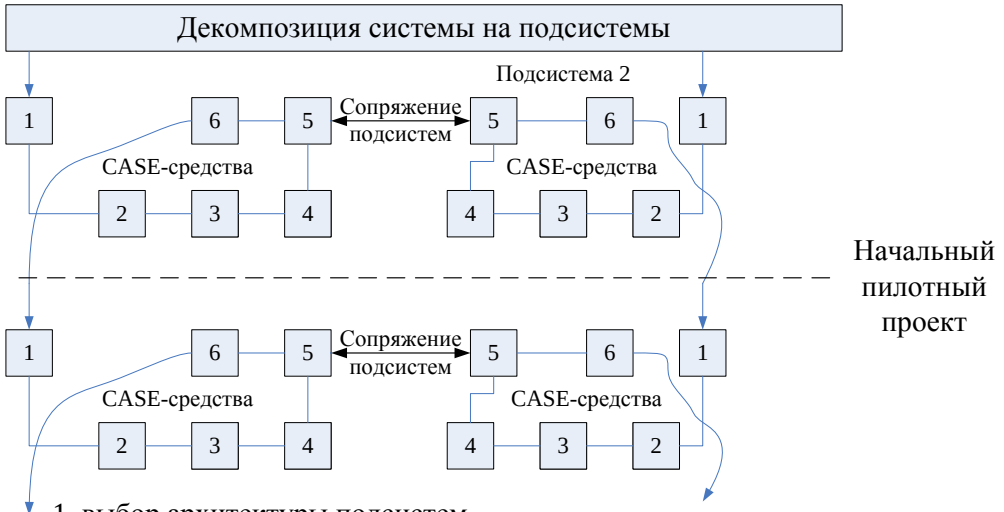
Спиральная модель

Для преодоления перечисленных проблем была предложена спиральная модель ЖЦ делающая упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача – как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.



Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.



1. выбор архитектуры подсистем
2. выявление информационных потребностей конечных пользователей системы
3. концептуальное проектирование
4. логическое проектирование
5. отладка подсистем
6. сопровождение подсистемы

Каждый этап реализуется CASE-средствами. Содержание этапов совпадает с аналогичными в каскадной модели, но в отличие от нее, этапы реализуются с помощью CASE-средств (Computer Aided Software System Design) **– (автоматизированная технология создания программных информационных систем)** – использование этих средств позволяет существенно снизить время реализации витка спирали проектирования

подсистем (в этом состоит основное преимущество спиральной модели), но это профессиональные средства, непредназначенные для конечных пользователей.

С помощью CASE-средств можно быстро сгенерировать проект хотя бы на уровне экранных форм и показать его конечному пользователю, который высказывает свои предложения и замечания, и на следующем витке реализуются они. Когда спецификация на уровне экранных форм будет будет согласована, то проектировщик может начинать детальную реализацию. Описанная схема разработки называют визуальным проектированием.

Витков может быть много, разделение на этапы условно. Работы могут выполняться параллельно или в комплексной итерации на любом витке спирали.

Проектирование как конструирование

Идея конструирования информационных систем (ИС) из компонентов развивается на протяжении многих лет. По существу, речь идет о крупноблочном конструировании систем (Д. Видерхольд). Согласно этой идее, целесообразнее осуществлять капиталовложения в создание компонентов, которые можно было бы многократно использовать, чем всякий раз осуществлять разработку ИС сверху – вниз: от спецификаций требований к работающей системе.

Во-первых, следует отметить то обстоятельство, что компонентно-ориентированный подход к проектированию и реализации программных систем и комплексов является в некотором смысле развитием объектно-ориентированного и практически более пригоден для разработки крупных и распределенных систем (например, корпоративных приложений).

Прежде всего, сформулируем основополагающее для рассматриваемого подхода определение компонента. Под компонентом будем далее иметь в виду независимый модуль программного кода, предназначенный для повторного использования и развертывания. Как видно из определения, применение компонентного программирования призвано обеспечить более простую, быструю и прямолинейную процедуру первоначальной инсталляции прикладного программного обеспечения, а также увеличить процент повторного использования кода, т.е. усилить основные преимущества ООП.

Говоря о свойствах компонентов, следует прежде всего отметить, что это существенно более крупные единицы, чем объекты (в том смысле, что объект представляет собой конструкцию уровня языка программирования). Другими отличиями компонентов от традиционных объектов являются возможность содержать множественные классы и (в большинстве случаев) независимость от языка программирования.

Естественным требованием к современным информационным системам является способность наращивания их возможностей за счет использования дополнительно разработанных (а еще лучше - уже существующих) программных компонентов. Для этого требуется обеспечение свойства, называемого интероперабельностью.

Интероперабельность программного обеспечения (функциональность программного обеспечения) - способность программного продукта выполнять набор функций, определенных в его внешнем описании и удовлетворяющих заданным или подразумеваемым потребностям пользователей.

Под этим понимается соблюдение определенных правил или привлечение дополнительных программных средств, обеспечивающих возможность взаимодействия независимо от разработанных программных модулей, подсистем или даже функционально завершенных программных систем.

Новейшие технологии обеспечивают техническую возможность интероперабельного использования как программных (OMG CORBA), так и информационных компонент (WWW).

Аспекты описания систем

Наряду с декомпозицией описаний на иерархические уровни применяют разделение представлений о проектируемых объектах на аспекты.

Расчленение (декомпозиция) описаний по характеру отображаемых свойств объекта приводит к появлению ряда аспектов описания.

Аспект описания (страта) — описание системы или ее части с некоторой оговоренной точки зрения, определяемой функциональными, физическими или иного типа отношениями между свойствами и элементами.

Различают функциональный, информационный, структурный и поведенческий (процессный) аспекты.

Функциональное описание относят к функциям системы и чаще всего представляют его функциональными схемами.

Информационное описание включает в себя основные понятия предметной области (сущности), словесное пояснение или числовые значения характеристик (атрибутов) используемых объектов, а также описание связей между этими понятиями и характеристиками.

Информационные модели можно представлять графически (графы, диаграммы сущность - отношение), в виде таблиц или списков.

Структурное описание относится к морфологии системы, характеризует составные части системы и их межсоединения и может быть представлено структурными схемами, а также различного рода конструкторской документацией.

Поведенческое описание характеризует процессы функционирования (алгоритмы) системы и (или) технологические процессы создания системы. Иногда аспекты описаний связывают с подсистемами, функционирование которых основано на различных физических процессах.

Отметим, что в общем случае выделение страт может быть неоднозначным. Так, помимо указанного подхода очевидна целесообразность выделения таких аспектов, как функциональное (разработка принципов действия, структурных, функциональных, принципиальных схем), конструкторское (определение форм и пространственного расположения компонентов изделий), алгоритмическое (разработка алгоритмов и программного обеспечения) и технологическое (разработка технологических процессов) проектирование систем.

Типовые проектные решения – шаблоны проектирования

Идея, на которой основывается применение шаблонов проектирования, заключается в том, чтобы выработать стандартизованный подход к представлению общих решений, пригодных для часто встречающихся ситуаций при разработке ПО. Такой подход имеет ряд преимуществ.

- Со временем можно получить каталог шаблонов. Это позволяет новичкам в разработке ПО более эффективно использовать многолетний опыт их предшественников.

- Все решения, принимаемые при проектировании ПО, снабжены формализованным описанием, позволяющим оценить как достоинства, так и недостатки того или иного решения. Стандартизованные шаблоны облегчают как для новичков, так и для экспертов в области разработки ПО понимание того, как влияют на архитектуру создаваемой программы те или иные решения.

- Шаблоны проектирования позволяют всем, кто их использует, "говорить на одном языке". Это, в свою очередь, облегчает взаимопонимание между разработчиками при выборе того или иного решения. Вместо того чтобы детально описывать какое-либо архитектурное решение, мы говорим, какой шаблон мы намерены использовать.

Мы можем соотносить шаблоны друг с другом, что позволяет разработчику увидеть, какие шаблоны могут использоваться в одном проекте.

Шаблоны проектирования предоставляют в наше распоряжение эффективный способ делиться опытом со всеми участниками сообщества объектно-ориентированного программирования. Независимо от того, каким языком программирования мы владеем (C++, Smalltalk или Java), и того, в какой области мы приобрели опыт проектирования ПО (Web-проекты, интеграция устаревших систем или заказной проект), мы можем накапливать свой собственный опыт и делиться им с другими разработчиками. Если же говорить о долгосрочной перспективе, то данный подход позволяет улучшить состояние дел во всей индустрии разработки ПО.

История применения шаблонов

Идейным отцом применения шаблонов проектирования при разработке ПО считают профессора архитектуры Университета Калифорнии в Беркли Кристофера Александера (Christofer Alexander). В конце 70-х годов прошлого века он опубликовал несколько книг, в которых изложил основные принципы применения шаблонов в архитектуре, а также поместил каталог архитектурных шаблонов.

Именно посвященные шаблонам работы Александера и привлекли внимание к проблеме программистов, интересующихся объектно-ориентированным программированием (ООП). В их среде появились пионеры применения шаблонов в разработке ПО, которые на протяжении последующих десяти лет сформулировали основные принципы этого метода. Среди первопроходцев были Кент Бэк (Kent Beck) и Уард Каннингхэм (Ward Cunningham). В 1987 году на конференции OOPSIA (Object Oriented Programming, Systems, and Applications), посвященной ООП, прозвучал их доклад о применении шаблонов проектирования в языке Smalltalk. Еще одним адептом нового подхода стал Джеймс Коплайен (James Coplien), который в начале 90-х гг написал книгу о применении идиом (т.е. шаблонов) при разработке ПО на языке C++.

Ежегодные конференции OOPSLA сослужили хорошую службу для роста сторонников технологии шаблонов, так как на ее мероприятиях энтузиасты могли свободно делиться своими идеями со множеством благодарных слушателей. Кроме того, важную роль в

становлении данного направления технологии разработки ПО сыграли конференции некоммерческой организации Hillside Group, создателями которой были Кент Бэк и Гради Буч (Grady Booch).

Однако по-настоящему ощутимый вклад в дело популяризации технологии шаблонов проектирования внесла изданная в 1995 году книга *Design Patterns: Elements of Reusable Object-Oriented Software.* Ее авторы Эрик Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides) приложили так много усилий для распространения своих идей, что заслужили шутливое прозвище "Банда четырех" (GoF — gang of four). В книге представлено введение в довольно сложный язык шаблонов с иллюстрациями реализации обсуждаемых шаблонов на языке C++.

В это же время началось бурное развитие технологии Java, поэтому неудивительно, что Java-разработчики с самого начала стали применять шаблоны проектирования в своих проектах. Рост популярности технологии шаблонов проектирования среди Java-разработчиков проявился как в виде специальных презентаций на конференциях, таких как JavaOne, так и в появлении отдельной рубрики по шаблонам проектирования практически во всех специализированных журналах по программированию на Java.

Основные концепции технологии шаблонов

В центре технологии шаблонов лежит идея стандартизации информации о типичной проблеме и о методах ее решения. Один из самых полезных результатов, полученных в работе Александера, состоит в выделении некоторого способа представления шаблонов, который впоследствии был назван *формой* (form), или *форматом* (format). В форме Александера применялось пять направлений, по которым формализовалось обсуждение шаблонов и их применение в конкретных ситуациях.

Самое главное преимущество такого подхода состоит в том, что даже само название шаблона представляет собой ответ на вопрос: "Что можно сделать с помощью этого шаблона?" Кроме того, в форме содержится: описание проблемной области; пояснение того, как данный шаблон решает означенную проблему; в чем заключаются преимущества, недостатки и, возможно, компромиссы при использовании данного шаблона.

Естественно, когда шаблоны были восприняты миром ООП, появились вариации формы Александера, призванные учитывать интересы разработчиков ПО. Большинство из существующих сегодня форм были построены на одной или двух формах, получивших название "Канонических", или форм "Четверки". В данной книге используется одна из вариаций форм "Четверки", согласно которой при представлении шаблонов освещаются следующие вопросы.

- *Название (Name).* Слово или выражение, описывающее основное назначение шаблона.
- *Также известен как (Also Known As).* Альтернативные названия (если, конечно, таковые имеются).
- *Свойства шаблона (Pattern Properties).* Классификация шаблона. Мы будем определять шаблон, используя термины из двух следующих групп.

Тип:

- *Производящие шаблоны (Creational patterns),* предназначенные для создания объектов.

- *Поведенческие шаблоны (Behavioral patterns),* обеспечивающие координацию функционального взаимодействия между объектами.
- *Структурные шаблоны (Structural patterns)*, используемые для управления статическими, структурными связями между объектами.
- *Системные шаблоны (System patterns)*, предназначенные для управления взаимодействием на системном уровне.

Уровень:

- *Отдельный класс (single class).* Шаблон применяется к отдельному, независимому классу.
- *Компонент (component).* Шаблон используется для создания группы классов.
- *Архитектурный (architectural).* Шаблон используется для координации работы систем и подсистем.

Повторное использование и абстракция программного кода

Технология шаблонов проектирования стала важным эволюционным этапом в развитии таких концепций, как *абстракция (abstraction)* и *повторное использование (reuse)* программного кода. Обе эти концепции занимают центральное место в самой идее программирования (некоторые даже полагают, что они являются важнейшими концепциями).

Абстракция — это метод, с помощью кот`орого разработчики могут решать сложные проблемы, последовательно разбивая их на более простые. Затем можно использовать имеющиеся готовые решения полученных типовых простых проблем в качестве строительных блоков, из которых разработчики получают решения, пригодные для реализации повседневных сложных проектов.

В ходе развития объектно-ориентированных языков программирования был совершен огромный скачок вперед в области абстракции и повторного использования программного кода. С помощью этой технологии была создана целая плеяда высокопроизводительных методов его создания.

Например, чего стоит только одна концепция класса как "эталона" объектов, с ее объединением двух ранее возникших механизмов: функциональной абстракции и абстракции данных. Объединяя структуру сущности (данные) с функциональностью, которая применяется к сущности (поведение), мы получаем эффективный метод повторного использования программного элемента.

Помимо основополагающего понятия класса, объектно-ориентированные языки принесли с собой множество других методов использования существующего кода.

В качестве примера можно привести такие концепции, как концепции подклассов и интерфейсов, открывших совершенно новые возможности в повторном использовании программного кода при разработке ПО.

Метод	*Повторное использование*	*Абстракция*	*Универсальность подхода*
Копирование и вставка	Очень плохо	Отсутствует	Очень плохо
Структуры данных	Хорошо	Тип данных	Средне — хорошо
Функциональность	Хорошо	Метод	Средне — хорошо
Типовые блоки кода	Хорошо	Типизируемая операция	Хорошо
Алгоритмы	Хорошо	Формула	Хорошо
Классы	Хорошо	Данные + метод	Хорошо
Библиотеки	Хорошо	Функции	Хорошо — очень хорошо
API	Хорошо	Классы утилит	Хорошо — очень хорошо
Компоненты	Хорошо	Группы классов	Хорошо — очень хорошо
Шаблоны проектирования	Отлично	Решения проблем	Очень хорошо

В столбце "Абстракция" таблицы указаны сущности, для которых выполняется абстракция, а в столбце "Универсальность подхода" показано то, насколько легко применить соответствующий метод, не прибегая к изменениям или переработке кода. Обратите внимание на то, что показатель степени повторного использования очень сильно зависит от эффективности применения того или иного метода на практике. Это понятно, так как любая, даже самая лучшая возможность может быть использована как правильно, так и неправильно.

Одной из самых выдающихся возможностей, предоставляемых шаблонами проектирования, является то, что они позволяют разработчикам более эффективно применять другие методы повторного использования программного кода. Шаблон в определенных ситуациях может, например, использоваться как руководство для эффективного управления наследованием или же для эффективного назначения связей между классами, обеспечивая тем самым решение той или иной проблемы.

Производящие шаблоны

Производящие шаблоны (creational patterns) предназначены для обеспечения выполнения одной из самых распространенных задач в ООП — создания объектов в системе.

В ходе работы большинства объектно-ориентированных систем, независимо от уровня их сложности, создается множество экземпляров объектов. Производящие шаблоны облегчают процесс создания объектов, предоставляя разработчику следующие возможности:

- Единый способ получения экземпляров объектов. В системе обеспечивается механизм создания объектов без необходимости идентификации определенных типов классов в программном коде.

- Простота. Некоторые из шаблонов упрощают процесс создания объектов до такой степени, что полностью избавляют разработчика от необходимости написания большого и сложного программного кода для получения экземпляра объекта.

- Учет ограничений при создании. Некоторые шаблоны позволяют при создании объектов налагать ограничения на их тип или количество в соответствии с установленными требованиями системы.

Основные производящие шаблоны проектирования.

- Abstract Factory. Обеспечивает создание семейств взаимосвязанных или зависящих друг от друга объектов без указания их конкретных классов.

- Builder. Упрощает создание сложных объектов путем определения класса, предназначенного для построения экземпляров другого класса. Шаблон Builder генерирует только одну сущность. Хотя эта сущность в свою очередь может содержать более одного класса, но один из полученных классов всегда является главным.

- Factory Method. Определяет стандартный метод создания объекта, не связанный с вызовом конструктора, оставляя решение о том, какой именно объект создавать, за подклассами.

- Prototype. Облегчает динамическое создание путем определения классов, объекты которых могут создавать собственные дубликаты.

- Singleton. Обеспечивает наличие в системе только одного экземпляра заданного класса, позволяя другим классам получать доступ к этому экземпляру.

Два из перечисленных выше шаблона, а именно Abstract Factory и Factory Method, базируются исключительно на концепции определения создания настраиваемых объектов. Подразумевается, что разработчик, применяющий эти шаблоны, при реализации системы обеспечит механизм расширения создаваемых классов или интерфейсов. В силу этой особенности данные шаблоны часто объединяются с другими производящими шаблонами.

Поведенческие шаблоны

Поведенческие шаблоны (behavioral patterns) применяются для передачи управления в системе. Существуют методы организации управления, применение которых позволяет добиться значительного повышения как эффективности системы, так и удобства ее эксплуатации. Поведенческие шаблоны представляют собой проверенные на практике методы и обеспечивают понятные и простые в применении эвристические способы организации управления.

В данной главе рассматриваются следующие поведенческие шаблоны.

- Chain of Responsibility. Предназначен для организации в системе уровней ответственности. Использование этого шаблона позволяет установить, должно ли сообщение обрабатываться на том уровне, где оно было получено, или же оно должно передаваться для обработки другому объекту.

- Command. Обеспечивает обработку команды в виде объекта, что позволяет сохранять ее, передавать в качестве параметра методам, а также возвращать ее в виде результата, как и любой другой объект.

- Interpreter. Определяет интерпретатор некоторого языка.

- Iterator. Предоставляет единый метод последовательного доступа к элементам коллекции, не зависящий от самой коллекции и никак с ней не связанный.

- Mediator. Предназначен для упрощения взаимодействия объектов системы путем создания специального объекта, который управляет распределением сообщений между остальными объектами.

- Memento. Сохраняет "моментальный список" состояния объекта, позволяющий такому объекту вернуться к исходному состоянию, не раскрывая своего содержимого внешнему миру.

- Observer. Предоставляет компоненту возможность гибкой рассылки сообщений интересующим его получателям.

- State. Обеспечивает изменение поведения объекта во время выполнения программы.

- Strategy. Предназначен для определения группы классов, которые представляют собой набор возможных вариантов поведения. Это дает возможность гибко подключать те или иные наборы вариантов поведения во время работы приложения, меняя его функциональность "на ходу".

- Visitor. Обеспечивает простой и удобный в эксплуатации способ выполнения тех или иных операций для определенного семейства классов. Это достигается путем централизации с помощью данного шаблона возможных вариантов поведения, что позволяет модифицировать или расширять их, не затрагивая классы, на которые распространяются эти варианты поведения.

- Template Method. Предоставляет метод, который позволяет подклассам перекрывать части метода, не прибегая к их переписыванию.

Структурные шаблоны

Структурные шаблоны (structural patterns) с одинаковой эффективностью применяются как для разделения, так и для объединения элементов приложения. Способы воздействия структурных шаблонов на приложение могут быть самые разные.

Например, шаблон Adapter может обеспечить возможность двум несовместимым системам обмениваться информацией, тогда как шаблон Facade позволяет отобразить упрощенный пользовательский интерфейс, не удаляя ненужных конкретному пользователю элементов управления.

Структурные шаблоны, рассмотренные в данной главе, имеют следующее назначение.

- Adapter. Обеспечение взаимодействия двух классов путем преобразования интерфейса одного из них таким образом, чтобы им мог пользоваться другой класс.

- Bridge. Разделение сложного компонента на две независимые, но взаимосвязанные иерархические структуры: функциональную абстракцию и внутреннюю реализацию. Это облегчает изменение любого аспекта компонента.

- Composite. Предоставление гибкого механизма для создания иерархических древовидных структур произвольной сложности, элементы которых могут свободно взаимодействовать с единым интерфейсом.

- Decorator. Предоставление механизма для добавления или удаления функциональности компонентов без изменения их внешнего представления или функций.
- Facade. Создание упрощенного интерфейса для группы подсистем или сложной подсистемы.
- Flyweight. Уменьшение количества объектов системы с многочисленными низкоуровневыми особенностями путем совместного использования подобных объектов.
- Half-Object Plus Protocol (HOPP). Предоставление единой сущности, которая размещается в двух или более областях адресного пространства.
- Proxy. Представление другого объекта, обусловленное необходимостью обеспечения доступа или повышения скорости либо соображениями безопасности

Источники

1. Грекул – Проектирование ИС Лекция 2.
2. Стелтинг, Маассен – Шаблоны проектирования в Java (16-19)
   Производящие шаблоны (стр.25-26)
   Прототип (стр. 48-51)
   Поведенческие шаблоны (стр. 61-62)
   Итератор (стр. 88-90)
   Наблюдатель (стр. 111-113)
   Структурные шаблоны (стр. 155)
   Фасад (стр. 189-191)

Абстрактная фабрика

Порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы.

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Плюсы:

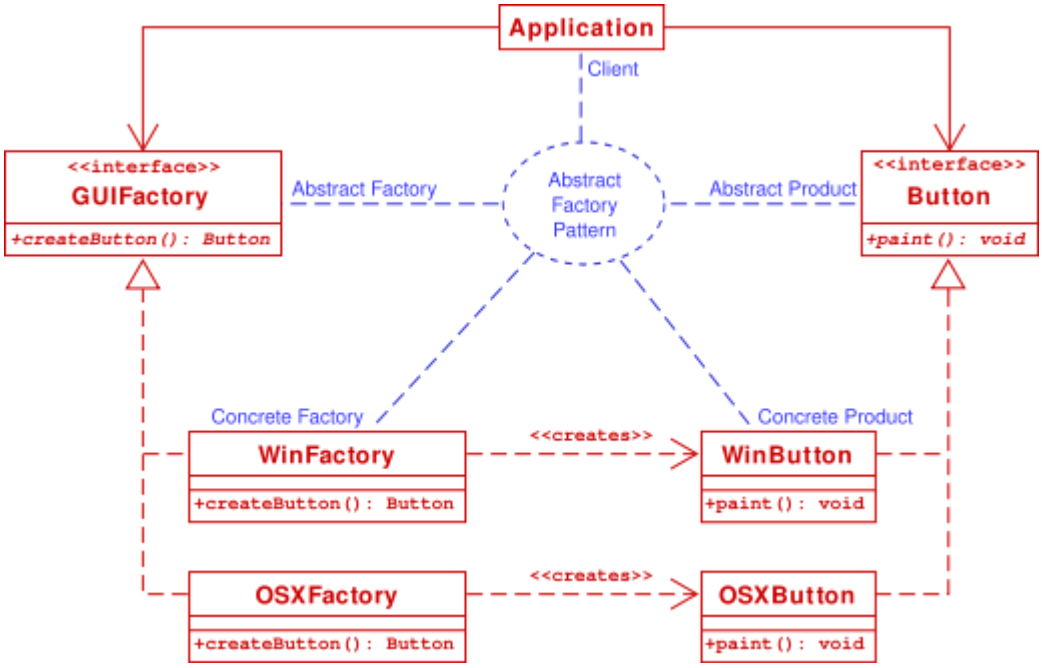
изолирует конкретные классы

упрощает замену семейств продуктов

гарантирует сочетаемость продуктов

Минусы:

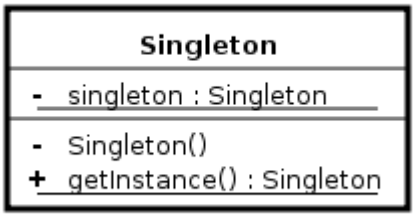
сложно добавить поддержку нового вида продуктов.



Синглтон

Объект-одиночка

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



Плюсы:

контролируемый доступ к единственному экземпляру

уменьшение числа имён

допускает уточнение операций и представления

допускает переменное число экземпляров

бóльшая гибкость, чем у операций класса

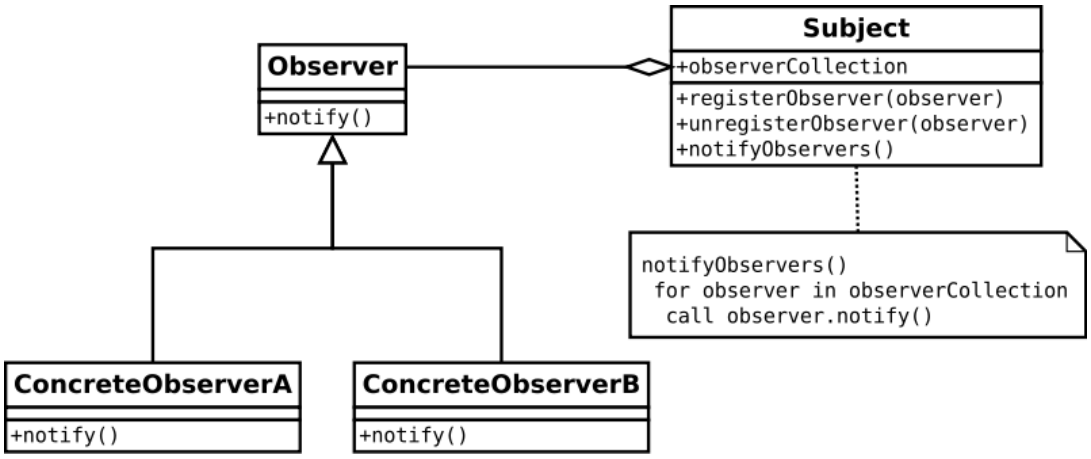
Минусы:

Глобальные объекты могут быть вредны для объектного программирования, в некоторых случаях приводя к созданию немасштабируемого проекта.

Усложняет написание модульных тестов

Наблюдатель

Поведенческий шаблон. Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.



Пример кода в отдельном файле observer.py

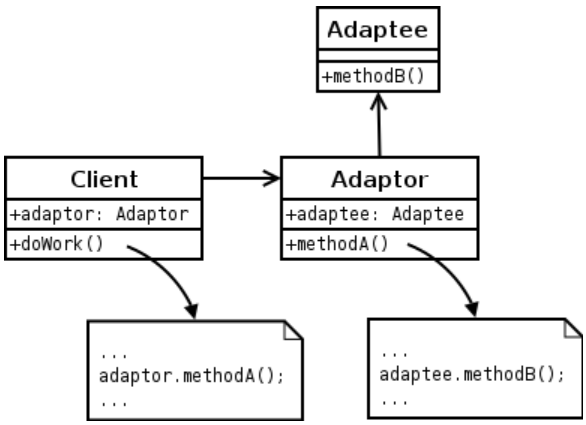
Адаптер

Структурный шаблон, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Задача: Система поддерживает требуемые данные и поведение, но имеет неподходящий

интерфейс.

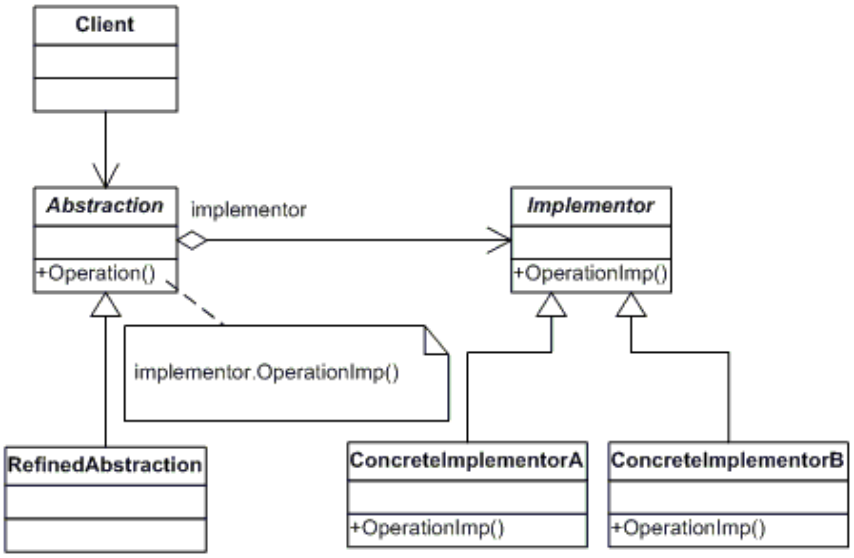
Решение:Адаптер предусматривает создание класса-оболочки с требуемым интерфейсом




```
class Adaptee:
    def specific_request(self):
        return 'Adaptee'
class Adapter:
    def __init__(self, adaptee):
        self.adaptee = adaptee
    def request(self):
        return self.adaptee.specific_request()
client = Adapter(Adaptee())
    print client.request()
```

Мост

Структурный шаблон, предназначенный для того, чтобы «разделять абстракцию и реализацию так, чтобы они могли изменяться независимо».



Пример в отдельном файле bridge.py.

Приложение 2

Сложность алгоритмов

Понятие сложности алгоритма входит в теорию алгортмов, которая изучает общие свойства и закономерности алгоритмов, а также разнообразные модели их представления.

Задачи теории алгоритмов:

- Асимптотический анализ сложности алгоритмов
- Доказательство асимптотической неразрешимости алгоритмов
- Классификация алгоритмов в соответствии с классами сложности

Понятие сложности также является одним из основных объектов изучения в теории сложности вычислений, изучающей стоимость работы, требуемой для решения вычислительной проблемы.

Стоимость обычно измеряется такими абстрактными понятиями, как *время* и *пространство*, называемыми вычислительными ресурсами.

Время – количество элементарных шагов, необходимых для решения задачи. Время является основным параметром, характеризующим быстродействие алгоритма. Оно также называется *вычислительной сложностью*. Обычно учитывается худшее, среднее и лучшее время алгоритма.

Количество элементарных операций, затраченных алгоритмом для решения конкретного экземпляра задачи, зависит не только от размера входных данных, но и от самих данных. Например, количество операций алгоритма сортировки вставками значительно меньше в случае, если входные данные уже отсортированы. Чтобы избежать подобных трудностей, рассматривают понятие временной сложности алгоритма в худшем случае.

Временная сложность алгоритма (в худшем случае) — это функция размера входных и выходных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера (верхний предел). Располагая этим значением, мы точно знаем, что для выполнения алгоритма не потребуется большее количество времени. В задачах, где размер выхода не превосходит или пропорционален размеру входа, можно рассматривать временную сложность как функцию размера только входных данных.

Пример: Поиск информации по БД-> наихудший случай будет тогда, когда информация в БД отсутствует.

Аналогично понятию временной сложности в худшем случае определяется понятие временная сложность алгоритма в наилучшем случае. Также рассматривают понятие среднее время работы алгоритма, то есть математическое ожидание времени работы алгоритма. Иногда говорят просто: «Временная сложность алгоритма» или «Время работы алгоритма», имея в виду временную сложность алгоритма в худшем, наилучшем или среднем случае (в зависимости от контекста).

Пространство – объем памяти или места на носителе данных. По аналогии с временной сложностью, определяют пространственную сложность алгоритма, только здесь говорят не о количестве элементарных операций, а об объёме используемой памяти.

При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.

Асимптотические обозначения

Обозначение	**Интуитивное объяснение**	**Определение**
$f(n) \in O(g($	*f* ограничена сверху функцией*g* (с точностью до постоянного множителя) асимптотически асимптотическая верхняя граница	$\exists(C>0), n_0 : \forall(n>n_0)\ \lvert f(n)\rvert \leq \lvert Cg(n)\rvert$ или $\exists(C>0), n_0 : \forall(n>n_0)\ f(n) \leq Cg(n)$
$f(n) \in \Omega(g($	*f* ограничена снизу функцией *g*(с точностью до постоянного множителя) асимптотически асимптотическая нижняя граница	$\exists(C>0), n_0 : \forall(n>n_0)\ \lvert Cg(n)\rvert \leq \lvert f(n)\rvert$
$f(n) \in \Theta(g($	*f* ограничена снизу и сверху функцией *g* асимптотически асимптотические верхняя и нижняя границы	$\exists(C, C'>0), n_0 : \forall(n>n_0)\ \lvert Cg(n)\rvert < \lvert f(n)\rvert < \lvert C'g(n)\rvert$
$f(n) \in o(g(n))$	*g* доминирует над *f*асимптотически	$\forall(C>0), \exists n_0 : \forall(n>n_0)\ \lvert f(n)\rvert < \lvert Cg(n)\rvert$
$f(n) \in \omega(g($	*f* доминирует над *g*асимптотически	$\forall(C>0), \exists n_0 : \forall(n>n_0)\ \lvert Cg(n)\rvert < \lvert f(n)\rvert$
$f(n) \sim g(n)$	*f* эквивалентна *g*асимптотически	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

Примеры

«пропылесосить ковер» требует время, линейно зависящее от его площади (Θ(A)), то есть на ковер, площадь которого больше в два раза, уйдет в два раза больше времени. Соответственно, при увеличении площади ковра в сто тысяч раз, объем работы увеличивается строго пропорционально в сто тысяч раз, и т. п.

«найти имя в телефонной книге» требует всего лишь время, логарифмически зависящее от количества записей (O(log2(n))), так как открыв книгу примерно в середине, мы уменьшаем размер «оставшейся проблемы» вдвое (за счет сортировки имен по алфавиту). Таким образом, в книге, толщиной в 1000 страниц, любое имя находится не больше чем за раз (открываний книги).

Примеры сложностей алгоритмов

Название	**Сложность**	**Пример**	**Пример алгоритмов**
Константное время	O(1)	10	Определение четности числа Взятие элемента массива/хэш
Логарифмическое время	O(log n)	Log n, log n^2	Бинарный поиск
Линейное время	O(n)	N	Поиск наименьшего значение в неотсортированном массиве
Linearithmic time	O(nlogn)		Наиболее быстрый сортировки сравнением(quicksort и др)
Квадратичное время	O(n^2)	n^2	Сортировка пузырьком, сортировка вставками
Кубическое время	O(n^3)	n^3	Перемножение матриц nxn (без спец. алгоритмов)
Экспоненциальное время	2^O(n)	1.1^n, 10^n	Решение задачи о коммивояжере, динамическое программирование

Сортировка пузырьком.

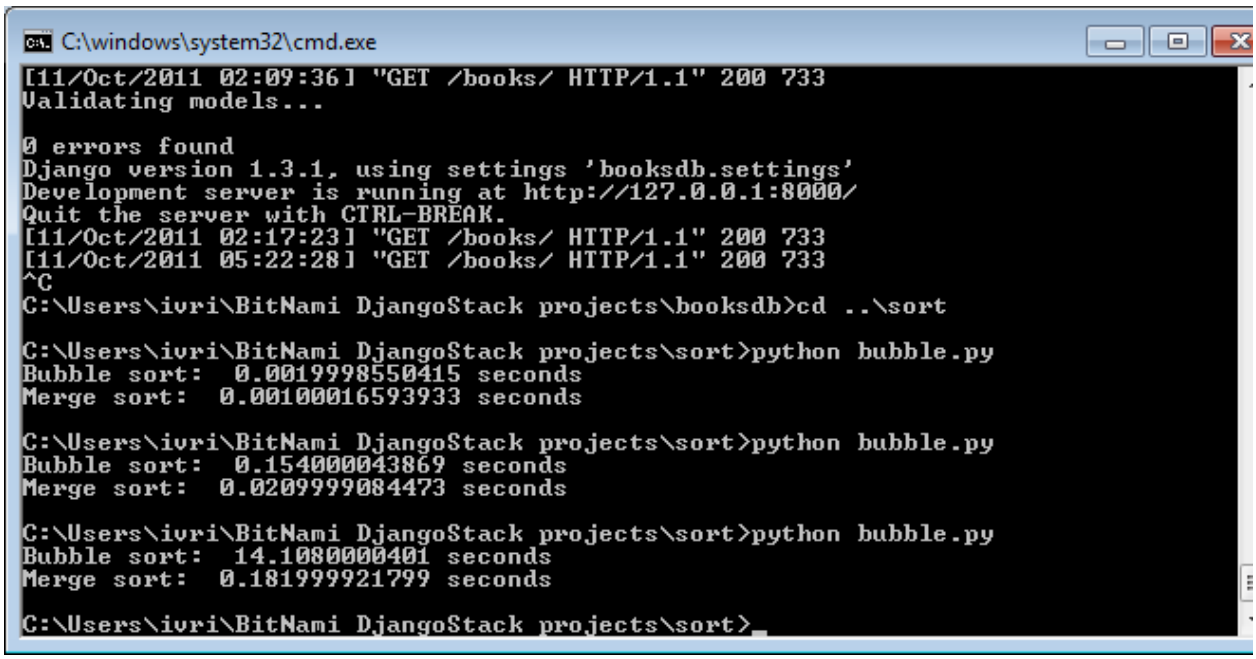
Временная сложность O(n^2):

```
def swap(arr, i, j):
        arr[i], arr[j] = arr[j], arr[i]
def bubble_sort(arr):
        i = len(arr)
        while i > 1:
                for j in xrange(i - 1):
                        if arr[j] > arr[j + 1]:
                                swap(arr, j, j + 1)
                i -= 1
```

Сортировка слияниями

Временная сложность O(n log n):

```
def mergesort(w):
 """Sort list w and return it."""
        if len(w)<2:
                return w
         else: mid=len(w)//2
# sort the two halves of list w recursively with mergesort and merge them
 return merge(mergesort(w[:mid]), mergesort(w[mid:]))
def merge(u, v):
"""Merge two sorted lists u and v together. Return the merged list r."""
        r=[]
        while u and v:
        # pop the smaller element from the front of u or v and append it to list r
                r.append( u.pop(0) if u[0] < v[0] else v.pop(0) )
        # extend result with the remaining end
                r.extend(u or v)
        return r
```



Выше приведено сравнение времени работы алгоритмов при сортировки 50, 500 и 5000 элементов (целочисленных) соответственно.

Код программы в отдельном файле bubble.py.