**Конспект**

**Критерии выбора языка программирования**

На данный момент существует огромное количество языков на любой, как говорится, вкус и цвет. Есть и компилируемые и интерпретируемые языки, есть очень простые и просто таки мозгодробительные языки, есть серьезные и шуточные языки. Много их, в общем, есть. И каждый из них создавался с определенной целью. Выбор языка программирования во время создания программы является очень важным моментом, от которого зависит очень и очень многое – скорость создания программы, скорость тестирования, возможность переноса на другие платформы, возможность быстрого внесения изменений, быстрота выполнения конечного продукта и так далее. При этом стоит помнить, что идеального языка не существует, все они обладают своими положительными и отрицательными качествами, которые будут так или иначе влиять на процесс разработки.

Итак, по каким же критериям выбирать язык для своего проекта?

1. Скорость работы конечного продукта.

Требовательным к скорости выполнения могут быть программы с большим объемом математических вычислений, например моделирование физических систем, расчеты большого объема экономических данных, выведение трехмерной графики и прочее. Для данных целей хорошо подойдут компилируемые языки: ассемблер, С/С++, фортран и т.д. Почему именно такие? После сборки программа не требует (грубо говоря) ничего лишнего и содержит в себе машинные команды, которые выполняются без лишних задержек. Схема работы таких программ такая: 1) программа исполняется сразу, так сказать она самодостаточна и не требует дополнительных библиотек; 2) программа кроме своего кода содержит вызовы библиотек с машинным кодом (как системных, так и входящих в проект), поэтому, кроме исполнения собственно своих команд, программа вызывает функции из библиотек; 3) в дополнение случаям 1 и 2, программа может работать через прослойку драйверов, которые написаны на языках низкого уровня и работают по умолчанию быстро. Как видно, максимум в схеме возможны 4 блока: программа -> библиотеки -> драйвера -> железо.

2. Объем занимаемой оперативной памяти.

Данное требование появляется, когда программа разрабатывается для встраиваемых систем, мобильных платформ, микроконтроллеров и так далее. В данных случаях, чем меньше памяти расходует программа на данном языке – тем лучше. К таким языкам, опять же, относятся ассемблер, С/С++, Objective-C и другие. Список языков подобен списку пункта 1, так как чем меньше функциональных блоков в схеме исполнения, тем меньше занимается и памяти компьютера.

Если данное требование некритично, то можно использовать «истинно высокоуровневые языки».

3. Скорость разработки программы.

Данное требование возникает тогда, когда начальник говорит «программа нужна не позже, чем вчера!» или еще какая срочность. Тогда выбор падает на высокоуровневые языки с максимально человеколюбивым синтаксисом. Это, например, Java, Flash и подобные. На данных языках время разработки может существенно сокращаться из-за обилия сторонних библиотек, максимально «очеловеченного» синтаксиса, и подобных вещей. Скорость выполнения программ, написанных на данных языках страдает, причем порой весьма ощутимо. Схема выполнения на примере Java:

Программа в виде байт-кода -> виртуальная машина-анализатор -> системные библиотеки -> драйвера -> железо.

Самым медленно работающим блоком в данной схеме является анализатор – он должен байт-код программы транслировать «на лету» в машинный код, при этом потратив много времени на точное определение инструкции. Поэтому быстрая разработка зачастую означает медленное выполнение.

4. Ориентированность на компьютер или человека

С кем будет работать программа в первую очередь? С человеком, или с компьютером? В пером случае программа должна обладать мощной графической частью, отвечающей требованиям дизайна и юзабилити. Разработка графической части зачастую требует достаточно много времени, т.к. отличается немалой сложностью. Здесь сложность возникает в том, что вывод графики – это немало математики, а значит присутствует требовательность к скорости исполнения, а из-за сложности разработки присутствует необходимость в высокоуровневом языке. В данном случае, на мой взгляд, очень хорошо подходит С++/C# с их одновременной и высокоуровневостью, и скоростью выполнения программ на них. Однако, если ГИП не очень сложный, но красивый, возможно использование Java и Flash, на которых создание красивых интерфейс гораздо проще, нежели на С++ и, тем более, С. Если программа ориентирована в первую очередь на «скрытую работу» с минимумом взаимодействия с пользователем, тогда выбор должен ложиться в сторону быстрых языков (ASM, C)

5. Кроссплатформенность.

Кроссплатформенность – возможность работы программы на различных платформах, в различных ОС с минимальными изменениями. В этой сфере можно выделить такие языки: Java, C#,Flash,C++ с различными библиотеками и другие, менее используемые, языки.

Java создавался с тем условием, что программы на данном языке должны работать на любой платформе, где есть JVM – Java Virtual Machine. Программы на Java вообще не требуют никаких изменений – после компиляции получается .jar файл, который будет работать и на Windows, и на Mac OS, и на Linux и еще немало где. Аналогичная ситуация и с Flash, только список платформ гораздо менее обширный. С С++ дело обстоит труднее. На чистом С++ написать кроссплатформенную программу довольно трудно, у кода возникает обширная избыточность, теряется достоинство в скорости выполнения. Облегчают задачу кроссплатформенные библиотеки, например, Qt, которые позволяют добиться принципа «один код на все платформы», однако на каждую платформу нужно программу собирать отдельно (при этом разными компиляторами).

В этот раздел так же можно включить интерпретируемые, скриптовые языки – для их работы нужно наличие интерпретатора языка в системе. Данные языки очень удобны в плане разработки, но достаточно медлительны. Схема их работы напоминает схему работы Java/Flash, только анализатор стал еще медленнее – полумашинный байт код анализировать «на лету» гораздо проще, чем человеческий код. Так же, это влечет к большему потреблению памяти.

6. Скорость внесения изменений, скорость тестирования

Проект стремительно развивается, в него постоянно вносятся изменения, порой немало? Тогда выбор должен падать на высокоуровневые языки, где любой функциональный блок можно быстро переписать. Для подтверждения – я думаю, гораздо проще дебажить тот же С++, чем ассемблер. А еще проще Java. Но тут очень много тонкостей, которые таятся даже не сколько в языке, сколько в разработчике с его стилем программирования и компиляторах. Тем не менее, язык вносит свою долю в это дело, так или иначе упрощая/осложняя работу программиста.

**Парадигмы программирования**

Паради́гма программи́рования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм (мультипарадигмальное программирование). Так, на языке Си, который не является объектно-ориентированным, можно работать в соответствии с принципами объектно-ориентированного программирования, хотя это и сопряжено с определёнными сложностями; функциональное программирование можно применять при работе на любом императивном языке, в котором имеются функции, и т. д.

1. Императи́вное программи́рование — это парадигма программирования (стиль написания исходного кода компьютерной программы), для которой характерно следующее:

- в исходном коде программы записываются инструкции (команды);

- инструкции должны выполняться последовательно;

- данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;

- данные, полученные при выполнении инструкции, могут записываться в память.

Императивная программа похожа на приказы (англ. imperative — приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках, то есть представляют собой последовательность команд, которые должен выполнить процессор.

2. Декларати́вное программи́рование — парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляют собой проблема и ожидаемый результат. Противоположностью декларативного является императивное программирование, описывающее на том или ином уровне детализации, как решить задачу и представить результат. В общем и целом, декларативное программирование идёт от человека к машине, тогда как императивное — от машины к человеку. Как следствие, декларативные программы не используют понятия состояния, в частности, не содержат переменных и операторов присваивания, обеспечивается ссылочная прозрачность.

Наиболее близким к «чисто декларативному» программированию является написание исполнимых спецификаций. К подвидам декларативного программирования также зачастую относят функциональное и логическое программирование. Несмотря на то, что программы на таких языках нередко содержат алгоритмические составляющие, архитектура в императивном понимании (как нечто отдельное от кодирования) в них также отсутствует: схема программы является непосредственно частью исполняемого кода.

3. Структу́рное программи́рование — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков. Концептуализирована в конце 1960-х — начале 1970-х годов на фундаменте теоремы Бёма — Якопини, математически обосновывающей возможность структурной организации программ, и работы Эдсгера Дейкстры «О вреде оператора goto» (англ. Goto considered harmful).

В соответствии с парадигмой, любая программа, которая строится без использования оператора goto, состоит из трёх базовых управляющих конструкций: последовательность, ветвление, цикл; кроме того, используются подпрограммы. При этом разработка программы ведётся пошагово, методом «сверху вниз».

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственно, усложнения программного обеспечения. В 1970-е годы объёмы и сложность программ достигли такого уровня, что традиционная (неструктурированная) разработка программ перестала удовлетворять потребностям практики. Программы становились слишком сложными, чтобы их можно было нормально сопровождать. Поэтому потребовалась систематизация процесса разработки и структуры программ.

4. Процеду́рное программи́рование — программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты. Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

5. Мо́дульное программи́рование — это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определённым правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

6. Объе́ктно-ориенти́рованное программи́рование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Идеологически ООП — подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что, в свою очередь, особенно важно при реализации крупных проектов.

Управляемость для иерархических систем предполагает минимизацию избыточности данных (аналогичную нормализации) и их целостность, поэтому созданное удобно управляемым — будет и удобно пониматься. Таким образом, через тактическую задачу управляемости решается стратегическая задача — транслировать понимание задачи программистом в наиболее удобную для дальнейшего использования форму.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

- абстракция для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счёте — контекстное понимание предмета, формализуемое в виде класса;

- инкапсуляция для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;

- наследование для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя всё остальное, учтённое на предыдущих шагах;

- полиморфизм для определения точки, в которой единое управление лучше распараллелить или наоборот — собрать воедино.

То есть фактически речь идёт о прогрессирующей организации информации согласно первичным семантическим критериям: «важное/неважное», «ключевое/подробности», «родительское/дочернее», «единое/множественное». Прогрессирование, в частности, на последнем этапе даёт возможность перехода на следующий уровень детализации, что замыкает общий процесс.

7. Функциона́льное программи́рование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов — чистые функции).

8. Логи́ческое программи́рование — парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Самым известным языком логического программирования является Prolog.

Первым языком логического программирования был язык Planner, в котором была заложена возможность автоматического вывода результата из данных и заданных правил перебора вариантов (совокупность которых называлась планом). Planner использовался для того, чтобы понизить требования к вычислительным ресурсам (с помощью бэктрекинга — поиска с возвратом) и обеспечить возможность вывода фактов, без активного использования стека. Затем был разработан язык Prolog, который не требовал плана перебора вариантов и был, в этом смысле, упрощением языка Planner.

Из всех парадигм программирования используются структурное, процедурное, модульное, ООП.

**Правила оформления листинга программы**

Станда́рт оформле́ния ко́да (станда́рт коди́рования, стиль программи́рования) (англ. coding standards, coding convention или programming style) — набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения.

Стандарт оформления кода обычно принимается и используется некоторой группой разработчиков программного обеспечения для единообразного оформления совместно используемого кода. Целью принятия и использования стандарта является упрощение восприятия программного кода человеком, минимизация нагрузки на память и зрение при чтении программы.

Образцом для стандарта кодирования может стать набор соглашений, принятых в какой-либо распространённой печатной работе по языку (например, стандарт кодирования на языке Си, получивший сокращённое наименование K&R, происходит из классического описания Си его авторами — Керниганом и Ритчи), широко применяемая библиотека или API (так, на распространение венгерской нотации явно повлияло её использование в MS-DOS и Windows API, а большинство стандартов кодирования для Delphi используют, в той или иной мере, манеру кодирования библиотеки VCL).

Реже разработчик языка выпускает подробные рекомендации по кодированию. Например, выпущены стандарты кодирования на C# от Microsoft и на Java от Sun. Предложенная разработчиком или принятая в общеизвестных источниках манера кодирования в большей или меньшей степени дополняется и уточняется в корпоративных стандартах.