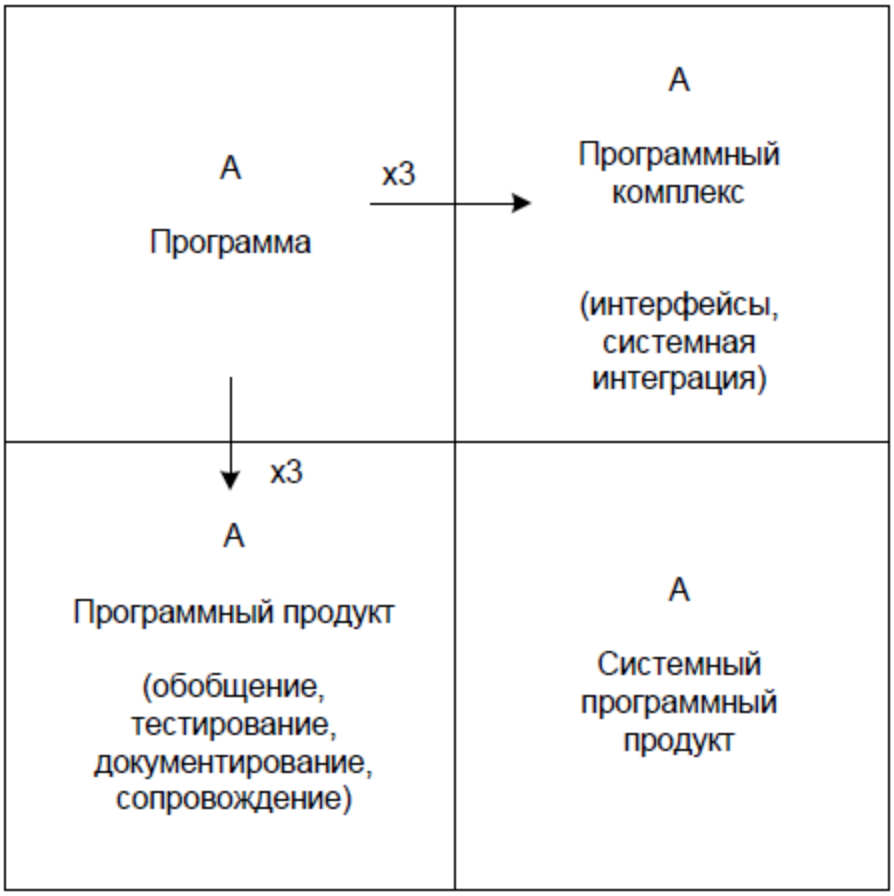
Сегодня мы поговорим о том, чем наш с вами полуторагодовой курс будет отличаться от аналогичных курсов, традиционно читаемых в других группах. Для начала давайте разберем, в чем заключаются понятия программы, программного продукта, программного комплекса и системного программного продукта.

Программа и программный продукт



[тут слова из Брукса]

Так вот, собственно традиционно в курсе “основы программирования” учат именно программировать, писать программы, т.е. левый верхний квадрат. Наш же курс более ориентирован на промышленное программирование. Как мы сегодня убедимся, жизнь программиста состоит не только из собственно программирования, и остальные активности (например, такие, как планирование и оценка, отладка и тестирование, версионирование и многое другое) не менее важны.

Модели разработки

Вообще, как вы думаете, как разрабатываются большие программные продукты?

Процесс создания и сопровождения систем чаще всего описывается в виде жизненного цикла (ЖЦ) ПО, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т.д. Такое формальное описание ЖЦ ПО позволяет спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом. Жизненный цикл ПО можно представить как ряд событий, происходящих с системой в процессе ее создания и использования.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данном ПО и заканчивая моментом его полного выхода из употребления. Модель жизненного цикла — структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

Фазы ЖЦ:

- возникновение и исследование идеи
- анализ и сбор требований (пилотный проект)
- планирование и проектирование
- разработка
- отладка и тестирование
- сдача
- сопровождение

Один очень важный вывод мы можем сделать даже при таком начальном знакомстве с понятием ЖЦП. Собственно программирование не является единственной деятельностью коллектива, занятого промышленными разработками. Более того, оно не является даже главным, наиболее трудоемким делом. Многие исследования отдают на фазу программирования не более 15-20% времени, затраченного на разработку (сопровождение вообще бесконечно). Может быть, эти цифры заставят вас задуматься о важности и других аспектов образования — от умения найти и обосновать эффективный алгоритм до искусства владения родным языком, как устным, так и письменным.

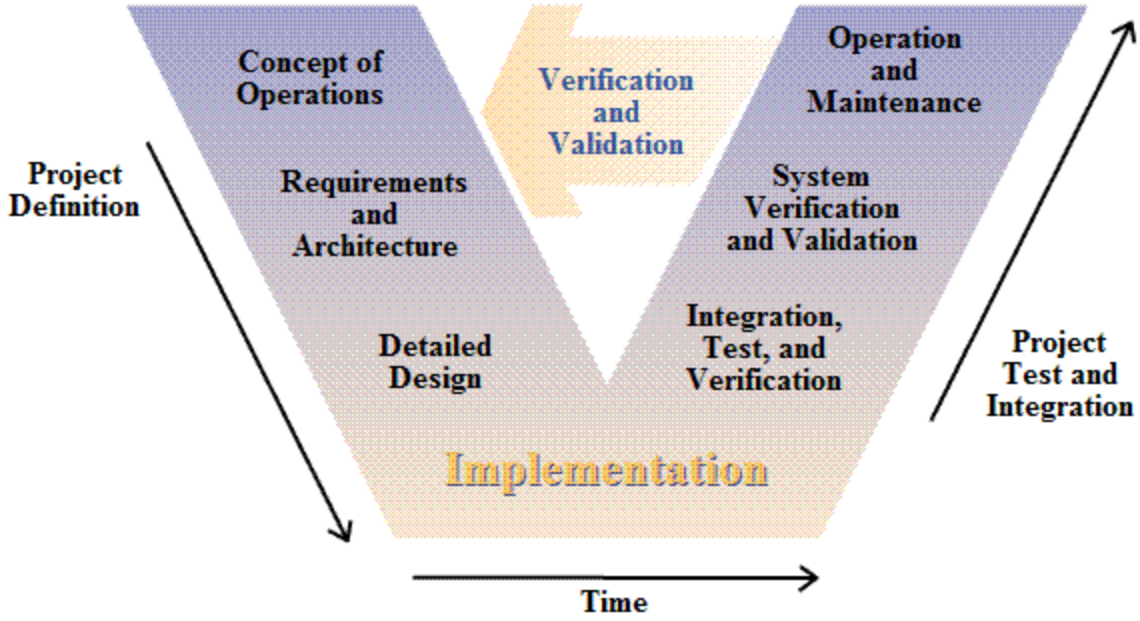
Существует несколько моделей того, как эти стадии организуются в рабочий процесс.

Водопадная модель:

1. Определение требований
2. Проектирование
3. Конструирование (также «реализация» либо «кодирование»)
4. Интеграция
5. Тестирование и отладка (также «верификация»)
6. Инсталляция
7. Поддержка

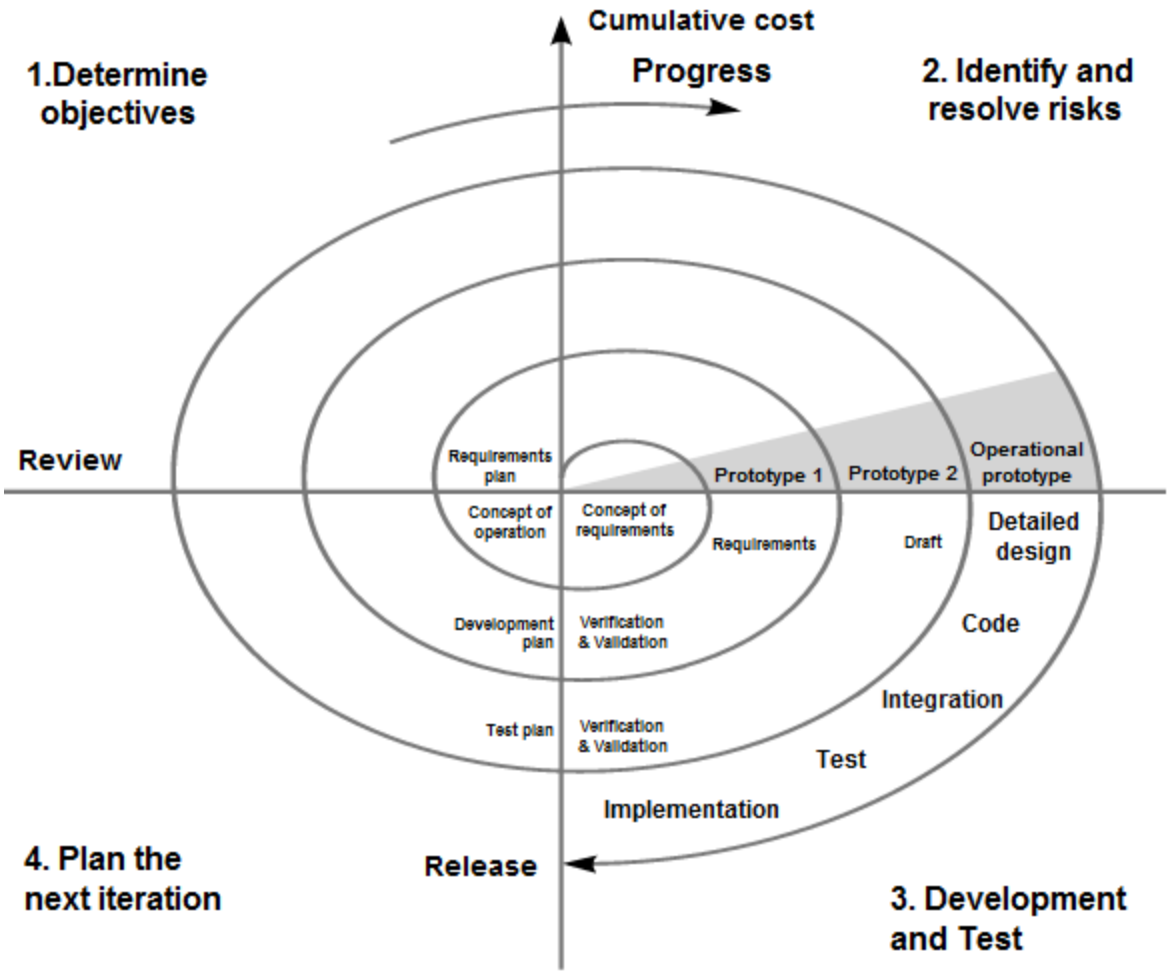
Все стадии идут строго друг за другом, возврата к предыдущим не допускается. Позаимствована из производства железа, где возврат к предыдущему этапу очень дорог, если вообще возможен. Почему хорошо: исправление ошибок в требованиях, оставленных незамеченными до фазы реализации или поддержки, будут стоить в 50-200 раз дороже, чем если бы их заметили на стадии анализа требований и проектирования.

V-образная модель:

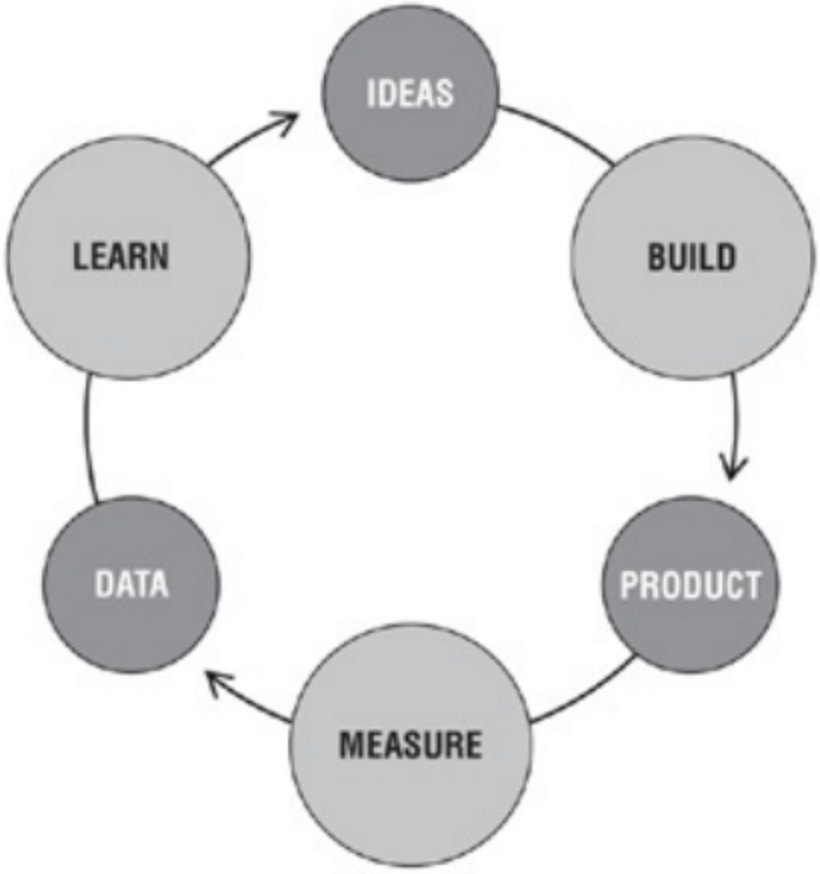


Спиральная модель:

- оценка и разрешение рисков,
- определение целей,
- разработка и тестирование,
- планирование.



Для проектов, в которых очень много неопределенности, хорошо подойдет следующая схема. Например, есть мнение, что она крайне полезна для компаний-стартапов, где постоянный хаос и все непрерывное меняется. Повторяя цикл раз за разом, можно формировать все более точную картину окружающего мира, пользователей, их желаний и, как следствие, создаваемого продукта.



Структурное проектирование

Подход к программированию, предложенный Эдсгером Дейкстрой в 70-х годах прошлого века как вариант преодоления того треша и угара, который творился в программировании в то время. Сейчас многое кажется естественным (особенно после того, как я вам каждому плешь в этом месте проел), однако раньше код писали как придется, не особо заботясь о структуре получаемых программ (в противовес структурному такой подход называют неструктурное программирование, сейчас же такой код называют спагетти-код). В результате получался фарш, в котором было куча безусловных переходов типа goto, весь код писался одним большим полотном и разобраться в нем было смерти подобно.

Дейкстра (и потом оформивший это все Н.Вирт) предложил следующее (например, много шума наделала его знаменитая статья о том, что в программировании можно обойтись без goto вовсе):

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:

- последовательное исполнение — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
- ветвление — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
- цикл — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

2. В программе базовые конструкции могут быть вложены друг в друга произвольным образом, но никаких других средств управления последовательностью выполнения операций не предусматривается. Повторяющиеся фрагменты программы (либо не повторяющиеся, но представляющие собой логически целостные блоки) могут оформляться в виде подпрограмм (процедур или функций). В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется инструкция вызова подпрограммы. При выполнении такой инструкции выполняется вызванная подпрограмма, после чего исполнение программы продолжается с инструкции, следующей за командой вызова подпрограммы.
3. Разработка программы ведётся пошагово, методом «сверху вниз» (так называемое, нисходящее проектирование).

Нисходящее проектирование заключается в том, что мы разбиваем большую задачу на меньшие подзадачи, которые выделяем в отдельные подпрограммы и рассматриваем порознь. Собственно, когда мы пишем код основной программы, мы вставляем в код вызовы функций, которые соответствуют подзадачам нашей текущей задачи. Для вызовов делаются самые простые реализации-”заглушки”, которые позволяют скомпилировать код, но ничего особо не делают. Это позволяет реализовать программу в виде таких вот крупных блоков, убедиться, что они вызываются в нужном порядке, передают друг другу данные и т.п. Этому помогает строгое описание проектировщиком входов и выходов фукнций и модулей.

Когда убедитесь, что все ок, заглушки по одной заменяются реальным кодом (разумеется, к каждой из заглушек может применяться тот же самый подход с декомпозицией на более мелкие задачи). Такая последовательность действий гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством участков кода (которые он может удержать в голове и не сойти с ума), и может быть уверен, что общая структура всех более высоких уровней программы верна, так что про них можно пока не думать вовсе.

Как только вы убедитесь, что некоторая часть задачи может быть реализована в виде отдельного модуля, постарайтесь больше не думать об этом, т. е. не уделяйте слишком много внимания тому, как именно он будет реализован.

Проектированию структуры данных следует уделять не меньше внимания, чем проектированию процессов или алгоритмов. Во многих случаях в состав этих данных входят требования к межмодульным интерфейсам, и проектирование этих модулей не продвинется существенно до тех пор, пока не будут тщательно описаны соответствующие интерфейсы.

Модули

Обычный для процедурного программирования подход к написанию больших программ — это разбиение их на модули. Модуль — это логически обособленный кусок функциональности программы, обладающий интерфейсом и реализацией. Интерфейс — это та функциональность, которую модуль предоставляет клиентам, реализация — это то, как модуль реализует эту функциональность. Клиенты модуля видят только интерфейс, и чем меньше они знают о реализации, тем лучше для них — это так называемая концепция сокрытия деталей реализации. Это удобно, поскольку позволяет менять реализацию модуля, не внося никаких изменений в код, который его использует. Пример модуля — модуль сортировки или работы со списками, которые вы делали в домашке. Например, интерфейс модуля сортировки — объявление функции qsort(), реализация — реализация функции qsort() и функций, которые нужны, чтобы qsort() работала. Определение разумного интерфейса модуля — такого, чтобы с одной стороны, предоставить максимум возможностей клиентам, и с другой стороны, не раскрывать как можно больше деталей реализации — сложная задача проектирования.

- Надо проектировать программу в регулярном иерархическом виде.
- Должна проводиться четкая декомпозиция программы на модули.
- Модуль должен быть максимально минимизирован.
- Модуль не дает и не предполагает побочных эффектов.
- Модуль не зависит от реализации других модулей.
- Модуль реализует независимую (и по возможности единственную) функциональность.
- Модуль может быть отдельно запрограммирован и протестирован.
- Модуль спроектирован с учетом принципа сокрытия данных.
- Реализация модуля может быть изменена при сохранении интерфейсов
- В процессе разработки могут использовать функции-заглушки.
- Процесс разработки идет посредством последовательной детализация (на каждом этапе существует вариант программы, который можно тестировать).

Кто забыл, в C++ модуль представляет собой пару из .cpp и .h-файла, в .h-файле описывается интерфейс модуля, в соответствующем .cpp-файле описывается его реализация. #pragma once, все дела. Инклудить .cpp ни в коем случае не надо, линковщик от этого офигевает.

Про тестирование

В этом семестре про тестирование просто хочется сказать, что оно есть, благо что в следующем семестре (кто туда доучится) будет отдельная большая пара про тестирование.

Тестирование — это процесс выявления ошибок в ПО. Касаемо ошибок есть замечательные аксиомы, приписываемые разным авторам:

1. Любая программа содержит ошибки.
2. Если программа не содержит ошибок, их содержит алгоритм, который реализует

эта программа.

3. Если ни программа, ни алгоритм ошибок не содержат, такая программа даром никому не нужна.

К сожалению, это очень жизненно, поскольку так или иначе ошибаются все. Так вот тестирование — это именно способ найти эти ошибки (про исправление никакой речи не идет, исправлению ошибок посвящена деятельность, называемая отладкой). Причем важный момент состоит в том, что тестирование не может доказать отсутствие ошибок в программе. Либо мы находим ошибки, либо мы прохо искали, либо см. правило 3 выше. Стопроцентное отсутствие ошибок в программе доказать формально можно, но для этого нужно запустить программу столько раз, чтобы управление прошло по всем ветвям условий и циклов всевозможное число раз. Несложный анализ показывает, что для любой нетривиальной программы это совершенно нереально в разумное время, так что остается довольствоваться тестированием.

Программирование и тестирование — в некотором смысле противоположные деятельности. Программирование созидательно, оно приближает нас к великому и вечному, тестирование же дотошно и деструктивно (в многих крупных компаниях тестировщикам даже премии выплачивают в зависимости от числа найденных багов). Это приводит к мысли о том, что для того, чтобы быть хорошим программистом и чтобы быть хорошим тестировщиком нужны совершенно разные качества. И уж точно эти две должности совмещать человеку практически невозможно. Особенно тестирование не должен проводить автор кода — он сознательно или несознательно будет создавать тесты на варианты исполнения кода, которые он знает и которые он реализовывал. Тестировщих же должен быть абсолютно непредвзят и суров.

Видов тестирования чуть менее, чем дофига, мы про этом будем подробно в следующем семестре говорить. Сейчас же можно рассмотреть основные два — тестирование черным и белым ящиком. Различаются они тем, имеет ли тестировщик возможность получить какую-то информацию о внутреннем представлении тестируемого кода (например, есть ли у него исходники или автор кода под рукой). Если есть (тестирование белым ящиком), то и методы тестирования тут могут быть совершенно другие — например, можно просканировать код на предмет возможных переполнений буфера, а потом их нацеленно проверять. Когда же никаких знаний о системе внутри нет (тестирование черным ящиком), то остается только скармливать программе какие-то данные и смотреть, как она себя ведет. Это вроде как менее эффективно, зато дешево и сердито — можно посадить совсем неквалифицированных тестеров, которые будут за еду нажимать кнопки в программе, сравнивать с эталоном и протоколировать результат.

Полезное чтение

- Фредерик Брукс, Мифический человеко-месяц, или Как создаются программные системы
- Питер Гудлиф. Ремесло программиста. Практика написания хорошего кода

- Эндрю Хант. Программист-прагматик. Путь от подмастерья к мастеру
- Джоэл Спольски. Джоэл о программировании
- А.Н.Терехов. Технология программирования