Инструменти за разработка на софтуер

Компютърни науки, 3 курс**СЪДЪРЖАНИЕ**

1. Увод
2. Инструменти за статичен анализ на кода…………………………………………………………………….6
   1. JavaScript инструменти…………………………………………………………………………………………..6
      1. JSLint………………………………………………………………………………………………………………7
      2. JSHint……………………………………………………………………………………………………………..9
      3. JavaScript Lint……………………………………………………………………………………………….10
      4. Closure compiler………………………………………………………………………………………..…11
   2. Python инструменти……………………………………………………………………………………………..13
      1. PyLint……………………………………………………………………………………………………………13
      2. PyChecker…………………………………………………………………………………………………….14
      3. PyFlakes………………………………………………………………………………………………………..14
3. Инструменти за създаване на документация…………………………………………………………….16
   1. Как да пишем документация……………………………………………………………………………….16
   2. JavaDoc ………………………………………………………………………………………………………………….17
   3. VSdocman………………………………………………………………………………………………………….…..19
   4. Sphinx…………………………………………………………………………………………………………………….21
   5. Asciidoc………………………………………………………………………………………………………………….23
4. Инструменти за анализ на код……………………………………………………………………………………25
   1. FxCop……………………………………………………………………………………………………………………..25
      1. Генериран код…………………………………………………………………………………………….25
      2. Правила за анализ на кода ………………………………………………………………………..26
      3. Предупреждения и грешки при анализ на кода……………………………………….26
   2. Инструмента instr…………………………………………………………………………………………………27
      1. Code coverage…………………………………………………………………………………………….27
      2. Употреба……………………………………………………………………………………………………..28
      3. Инструмента instr………………………………………………………………………………………..30
   3. Code coverage collection……………………………………………………………………………………….31
      1. Инструмента VSPerfMon……………………………………………………………………………31
      2. Възможни пречки за успешен code coverage collection…………………………….31
   4. Инструмента VSPerfCmd……………………………………………………………………………………...32
   5. Code coverage – примерен скрипт ………………………………………………………………………32
   6. Инструмента VSPerfReport…………………………………………………………………………………..33
5. Билд системи и автоматизация на билда ………………………………………………………………… 36
   1. Същност на билд системата ………………………………………………………………………………..37
   2. Видове билд системи според предназначението ……………………………………………..37
   3. Основни компоненти на една билд система ……………………………………………………..39
   4. Същност на билд процеса ……………………………………………………………………………………41
   5. Видове билд според предназначението …………………………………………………………….42
   6. Управление на билд системата ……………………………………………………………………………42
   7. Автоматизация на билда ……………………………………………………………………………………..43
      1. Скриптирани билдове ……………………………………………………………………………….43
      2. Първоначалният инструмент – Make ………………………………………………………..44
      3. Наследниците на Make …………………………………………….……………………………….45
         1. GNU Autoconf …………………………………………………………………………………46
         2. CMake …………………………………………………………………………………………….46
      4. Пълна автоматизация на билда в среда на непрекъсната интеграция (Continuous Integration) ……………………………………………………………………………….46

Дълъг е пътят, който софтуерът и компютрите са изминали – от ранната зора на компютрите до днес. От сметачната машина на Чарлз Бабидж до съвременните компютри, хардуерът е извървял огромен път и продължава да върви стремително напред и нагоре, а със него и софтуера. Като цяло те се развиват неразделно и колкото по-мощен и бърз става хардуера, толкова по-сложен става софтуера и включва все повече и повече нови функционалности. А с увеличаването на мащаба на даден софтуерен проект, нараства и нуждата от средства, които да улеснят разработчиците в непростия процес на разработка .

Отдавна са отминали времената на програмите написани на асемблер. С появата на езиците от високо ниво, се появява и нуждата от инструменти, които да осъществят връзката между разбираемия от човека код и машината. А с тях се появяват нови и нови инструменти, които улесняват най-различни аспекти от процеса на разработка – тестване, анализиране на кода и намиране на потенциални проблеми, дебъгване, генератори на потребителски интерфейс, генератори на документация, контрол на ревизиите, интеграция, компилация и свързване, пакетиране и т.н. Някои от тях са незаменими, други са по-скоро въпрос на избор дали ще включим в текущия процес. Но едно е сигурно – добре подбраните софтуерни инструменти могат да изиграят голяма роля в развитието на проекта.

Ето защо сме подбрали няколко много интересни и важни инструменти, които е добре да познаваме – инструменти за статичен анализ на код, за генериране на документация, за анализ на изпълнението и софтуерни билд системи.

Статичният анализ има за цел да открие синтактични грешки и потенциални проблеми в кода на програмата. Включването му в в процеса на разработка на софтуера е силно препоръчително. Ползите от анализа се отнасят не само до ранното откриване на грешки в кода и предпазването от потенциални проблеми в бъдеще, а и до ползите, които има за програмиста. Резултатите от анализа имат положителен резултат върху навиците на разработчиците и помага за привикването към писане на качествен код, а и липсата на грешки може да бъде добър психологически стимул.  
 Изборът на подходящ инструмент за целта е не по-маловажен проблем. Различните инструменти имат своите плюсове и минуси, които ги правят по-подходящи в дадена ситуация. Необходимо е и да се вземе предвид нивото на програмистите, които ще го използват, тъй като отделните инструменти предлагат различен подход на работа. Някои са изключително гъвкави и дават много голяма свобода на разработчика да избере най-подходящите настройки, докато други са доста по-консервативни и праволинейни и налагат единствен стил на писане, който се счита за добър стил. В някои случаи, общоприетите за лоши практики в програмирането могат да са подходящи и да са използвани напълно съсзнателно, въпреки опасностите, които крият.

Софтуерната документация или още документацията към изходния код представлява текст, който се счита за важна част от софтуера. Тя обяснява как работи дадена програма и как се ползва. При създаването на софтуерен продукт, кодът сам по себе си не е достатъчен. Нужно е да съществува текст към него, който да описва различните аспекти на неговото предназначение. От голямо значение е тази документация, да бъде лесна за поддържане от всички разработчици и тестери. Тя може да бъде няколко вида в зависимост от крайните потребители.

Съществуват различни инструменти за създаване на документация, които могат да бъдат използвани за автоматично генериране на документация – извличат коментари от самия сорс код и създават ръководства под различни форми, които позволяват на потребителя лесното намиране на функция или клас.

В днешния свят софтуерната сигурност е първото и най-важно изискване на всеки потребител когато се разработва ново приложение. Може да се разработи сигурно приложение като се намалят всички рискове още в процеса на цикъл на разработка. Най – добрия начин за намаляване на софтуерните рискове е да се правят преглеждания на кода и анализ на кода по време на цикъла на разработка.

Друг положителен аспект на анализа на кода е лесната идентификация на бъгове рано в цикъла на разработка преди тези бъгове да бъдят открити от тестващия екип или дори от потребителите на приложението. Идентификацията и поправката на критични бъгове рано в цикъла на разработка намалява рискове като намаляване продуктивността на разработчиците, създава ненужни трудни случаи при разработката, увеличава цената на разработката на софтуера и т.н.

Макар и изключително важна, билд системата е една от най-подценените части от процеса на разработка. Много от компаниите просто я пренебрегват и не се опитват да подобрят билд системата си, защото обикновено са експерти в коренно различна област (телекомуникации, компютърни игри, бизнес приложения и т.н) и техният ентусиазъм е насочен в друга посока. Съответно не се появяват задачи за подобрение и разработване на билд системата. А разрастването на продукта често води до усложняване на билд процеса.

Не е учудващо, че в една анкета в софтуерната индустрия, разработчиците са отбелязали средна загуба на продуктивност от 12% поради проблеми с билда, а част от участниците в анкетата споделят, че дори и 20-30% загуба поради същите проблеми не е рядко срещана. Тук трябва да отбележим, че анкетата е проведена в малки тимове до 20 души, където проблемите с мащабността на проекта са далеч по-малко засегнати. А при един голям проект, проблемите биха били далеч повече. Ето защо дизайнът на една добра билд система е ключът към по-добра продуктивност и ефективност на екипа и незаменима дългосрочна инвестиция в проекта, която ще ви спести и много главоболия.

**Инструменти за статичен анализ на кода**

Статичната анализ е анализ на компютърния софтуер, който се извършва без реално изпълнение на програмата. Инструментите за статичен анализ целят да проверят дали софтуерът отговаря на изискванията. Това става чрез проверка дали се спазват конвенциите при писане на код и откриване на лоши практики.

Статичният анализ изследва какво прави програмата без да я изпълнява чрез внимателно изпитване на изходния код. Статичният анализ може да диагностицира нарушения на правилата и конвенциите, които са необходими за правилното изпълнение на програмата или се предпочитат за някои нефункционални качествени показатели като например поддръжка и сложност. Също така може да съблюдава за съответствие със стандартите при писане на код и най-добрите практики за програмиране и безопасно програмиране.

Тъй като статичните инструменти за анализ не разчитат на тестове, които да симулират изпълнение на кода те могат да бъдат изключително задълбочени. Теоретично, статичните инструменти за проверка могат да изследват целия код, включително части от него, които са рядко изпълнявани на практика. Въпреки това, тъй като софтуерът действително не работи, могат да се генерират фалшиви положителни резултати. Това означава, че може да се докладват грешки, които не могат да появят в практиката.

Статичният анализ предоставя информация, която е валидна за всички възможни изпълнения на програмата. Тъй като за провеждането на проверката не е необходимо изпълнението на програмата, то тя може да бъде извършена преди имплементацията да е напълно завършена. Статичният анализ може да окаже се положителна и за екипа, провеждащ проверките, тъй като процесът помага на хората да осъзнаят необходимостта от спазването на стандартите.

В същото време, ранното откриване на грешки може да спести много време и средства, понеже е много по-лесно поправянето на грешки още в процеса на разработка, отколкото след интегрирането на всички модули на програмата или след пускането ѝ в употреба.

Инструментите за статичен анализ на кода, както и другите инструменти следящи за качеството на кода, са важни и полезни за JavaScript разработчиците и не бива да бъдат пренебрегвани.

Рано или късно количеството на кода нараства значително и наивни грешки, останали незабелязани в началото, могат да създадат проблеми и да затруднят процеса по отстраняването им. Инструментите за статичен анализ на кода помагат на разработчиците да открият тези проблеми още в зародиш.

Трябва да се има предвид, че докато инструментите за статичен анализ мога да забележат много различни видове грешки, те не са в състояние да докажат коректността на програмата или да открият „течове“ в паметта. За това винаги статичните средствата за анализ трябва да се комбинират с динамични тестове, които да следят за коректната работа на програмата и съответствието ѝ с функционалните изисквания.

Ще разгледаме няколко инструмента за статичен анализ на кода, които предоставят различни възможности за контрол от страна на потребителя.

**JSLint**

JSLint е инструмент за статичен анализ на кода използван в процеса на разработка на софтуер за проверка дали JavaScript кода е в съответствие с правилата за писане на код. Първоначално възниква като онлайн приложение, като вече има адаптирана версия и за изпълнение от командния ред.

JSLint е JavaScript програма, която търси проблеми в JavaScript програмите. Това е инструмент за качествен код. JSLint сканира JavaScript кода и, ако открие проблеми, връща съобщение описващо проблма и посочващо приблизителното му местоположение в изходния код. Проблемите не са непременно синтактични грешки, въпреки че те са най-голямата част. JSLint разглежда някои конвенции за стил на кода, както и структурни проблеми.

Липсата на съобщение за грешка не е доказателство за коректността на програмата. JSLint просто предлага възможност за откриване и локализиране на проблемите. JSLint би отхвърлил програми, въпреки че браузърите ще ги приемат и ще ги изпълнят, тъй като JSLint се занимава с качеството на кода (за разлика от браузърите).

Един от най-големите проблеми на JavaScript е зависимостта му от глобалните променливи, особено от подразбиращите се. Ако променлива не е изрично декларирана (чрез *var*), JavaScript предполага, че е глобална. Това може да прикрие сгрешени имена на променливи и да доведе до други проблеми.

JSLint очаква всички променливи и функции да бъдат декларирани преди да се използват или извикват. Това позволява откриването на подразбиращи се глобални променливи. Освен това, предварителното деклариране на променливи е добра практика, защото прави кода по-лесен за четене.

Понякога кода в JavaScript файл зависи от глобални променливи и функции декларирани в друг файл. Можем да упоменем това на JSLint чрез списък на тези променливи и функции предшествани от *var* в началото на файла.

Синтаксисът на JavaScript изисква използването на точка и запетая за отделянето на отделните изрази. JavaScript се опитва да направи тези точки и запетаи незадължителни чрез механизма за вмъкване на точка и запетая (semicolon insertion mechanism). Това е опасно, защото може да прикрие грешки. JSLint очаква всяко твърдение да завършва с точка и запетая, с изключение на функциите, *for*, *while*, *if*, *switch* и *try*. JSLint не очаква излишни точка и запетая или празни твърдения.

Операторът запетая усложнява изразите и може да маскира някои програмни грешки. JSLint предвижда изполването на запетая за разделител, но не и като оператор (освен при инициализиране на няколко променливи). Допълнителни запетаи не бива да се използват. Освен това не трябва да има запетая след посления елемент на масив или обект, защото тона може да бъде грешно изтълкувано от някои браузъри.

В много езици блоковете въвеждат обхват, така че променливите въведени в блока не се виждат извън него. В JavaScript това не е така. Само функциите имат обхват на променливите декларирани в тях – блока на функцията. Блоковете в JavaScript са в състояние да объркат и опитни програмисти и да доведат до неочаквани грешки, тъй като познатият синтаксис може да подведе програмиста относно обхвата на променливите. JSLint очаква задаване на блокове само с функции, *if*, *switch*, *while*, *do*, *for* и *try*.

В езиците с блоков обхват обикновено се препоръчва променливите да бъдат обявени на мястото на първата им употреба. Но тъй като JavaScript не разполага с блоков обхавт, по-добрият вариант е променливите на функцията да се декларират в горната ѝ част. Препоръчително е всички променливи във функцията да бъдат декларирани чрез *var* в началото на функцията.

JSLint очаква *if*, *while*, *do* и *for* винаги да бъдат с блокове обградени от фугурни скоби, дори и в случаите, когато JavaScript позволява пропускането им.

Често срещан проблем е пропускането на *break* след всеки *case* в *switch*. Това ще доведе до изпълнение на следващия *case*, дори и той да не отговаря на условията. За избягване на този проблем JSLint следи за наличието на *break*, *return* или *throw* след всеки *case*.

JavaScript позволява *var* дефиниции навсякъде в тялото на функцията. JSLint е по-строг в това отношение. Той очаква една единствена дефиниция чрез *var* и задължително преди използването на променливата, в началото на функцията.

Друго често срещана грешка, за която следи инструмента, е използването на *=* при задаване на *if*, *while* или *do*. Като цяло JSLint се стреми да наложи употребата на *===* и *==!* вместо на *==* и *=!*.

Също така JSLint следи за наличето на „недостъпен код“. За такъв се смята кодът непосредствено след *return*, *break*, *continue* или *throw* и, за това след тези оператори се очаква наличието на затваряща фигурна скоба, *case* или *default* (в *switch*).

Функцията *eval* осигурява достъп до компилатора на JavaScript. Макар понякога това да е необходимо, в повечето случаи е признак за лош стил и JSLint не толерира използването му.

В много C-подобни езици *void* е тип, но в JavaScript е префиксен оператор, койт винаги връща *undefined*. JSLint счита за грешка използването му, тъй като е ненужен и объркващ.

Конструкторите са функции, предназначени да се изпълняват с префикс *new*. *new* създава нов обект базиран на прототипа на функцията и свързва обекта към функционалния параметър *this*. Ако пренебрегнем използването на *new,* новият обект ще бъде създаден, но this ще бъде обвързан с глобални обект, а не с новосъздадения. Това е сериозна грешка.

JSLint налага конвенцията, че на конструктора трябва да се дават имена с главна първа буква. JSLint не очяква да срещне извикване на функция с главна първа буква, освен ако няма за префикс *new*. Съответно, използването на *new* при извикване на функция с малка първа буква ще се счита за грешка. Освен това JSLint не очаква изрази от вида на *new Number*, *new String* или *new Boolean* и препоръчва използването на *{}* и *[]* вместо *new Object* и *new Array*.

JSLint не прави проверка дали променливите използвани в кода са предварително дефинвирани. Причината е, че JavaScript задава стойност *undefined* по подразбиране на всички променливи. JSLint не следи и дали функциите извиквани чрез *new* са наистина конструктори (освен проверката за главна първа буква).

JSLint предоставя няколко опции, които контролират дейността и чувствителността му.

JSLint е наличен, освен като онлайн инструмент, и под още няколко форми. Той може да бъде използван и от командния ред или да бъде инсталиран като приставка за някои IDE-та и редактори като NetBeans, Visual Studio и Sublime.

В програмирането lint първоначално е името на програма, маркираща някои съмнителни конструкции, потенциални източници на грешки, в изходния код на C-програми. Терминът в момента се използва за инструменти съблюдаващи за евентуални промени в кода и за спазването на добрите практики при писане на редица езици за програмиране. Два от тези инструменти ще разгледаме сега – JSLint и Pylint. Но преди това ще се спрем на още един инструмент за JavaScript.

**JSHint**

JSHint е инструмент за откриване на грешки и потенциални проблеми в JavaScript кода и използването на конвенциите избрани от разработващия екип.

JSHint е изключително гъвкав, така че лесно може да се адаптира към специфичните насоки използвани от екипа програмисти.

JSHint се заражда като опит да се направи JSLint малко по-гъвкав, така че потребителите да могат сами да конфигурират инструмента, тъй като JSLint налага определен стил на кодиране, който не търпи промени. В последствие JSHint прераства в отделен инструмент за статичен анализ на кода със свои собствени цели и идеали.

Целта е да се помогне на JavaScript разработчиците да пишат сложни програми без да се притесняват от печатни грешки при въвеждането на кода.

JSHint сканира програмата и докладва за често срещани грешки и потенциални бъгове. Потенциалните проблеми могат да бъдат синтактични грешки, бъгове заради косвено преобразуване на типове или нещо друго.

JSHint може да се използва в различни среди за разработка, което помага за разпознаването на специфични за средата глобални променливи. JSHint е наличен за използване както в бръузъра, така и в ES5, Node.js и Rhino.

JSHint идва със стандартна конфигурация, но програмистът има възможност да промени голяма част от стандартните настройки. Има два основни начина за това. Може да се посочи конфигурационния файл ръчно чрез --*config* флага или да се използва стандартния *.jshintrc* файл. JSHint очаква да открие този файл в текущата работна директория, а в случай, че това не стане, ще продължи търсенето си нагоре в дървото на директориите, до самия корен. Това позволява да се задават различни конфигурационни файлове за всеки проект. Достатъчно е да се постави файла в основната директория на проекта, за да е достъпен от всички поддиректории. Конфигурационният файл е обикновен JSON файл, който определя кои опции на JSHint да са включени или изключени. Могат да се задават и специфични настройки за всеки отделен JavaScript файл. Това става директно в самия файл, като съответните опции се посочват като коментар в изходния код.

Има възможност и за задавен на списък с файлове, които JSHint да игнорира. Те се записват във файла *.jshintifnore*.

За улеснение опциите са разделени на два вида. Първият вид (enforcing) се използва, за да се направи JSHint „по-строг“, докато вторият вид (relaxing) се използва за потскане на някои предупреждения.

JSHint е наличен като плъгин за над 10 текстови редактори и IDE-та, сред които vim, Emacs, Visual Studio, Sublime Text 2, Eclips, Notepad++.

**JavaScript Lint**

JavaScript Lint е инструмент за статичен анализ на JavaScript кода. Трябва да се отбележи, че JavaScript Lint не е JSLint. Това са различни инструменти. JavaScript Lint следи за почти същите грешки, които споменахме за JSLint.

JavaScript Lint може да се изолзва както от командния ред, така и като плъгин за някои редактори и IDE-та като Visual Studio, Vim, Emacs и др.

JavaScript Lint позволява игнорирането на предупрежденията в част от кода или за целия файл. Това става чрез добавяне на коментар в JavaScript файла. Кодът между коментарите */\*jsl:ignore\*/* и */\*jsl:end\*/* ще бъде пренебрегнат при проверката за по-маловажни проблеми. За да наложим такова ограничение за целия файл е достатъчно да добавим */\*jsl:irnoreall\*/*. Важно е да се отбележи, че това ще изключи само предупрежденията, но не и грешките, които JavaScript Lint е преценил, че може да са фатални за изпълнението на програмата.

JavaScript Lint може по избор да проверява за променливи, функции и обект,и, които не съществуват, подобно на функцията на Visual Basic „option explicit“. С оглед да се направи JavaScript Lint достъпен за повече програмисти, тази мощна функция е изключена по подразбиране. Въпреки, че тя изисква повече работа за разбиране и прилагане, те осигурява по-високо ниво на защита срещу грешки в кода.

Променливи, които не са изрично декларирани, имат глобален обхват в JavaScript. Например, ако една функция използва променлива с име *X* и извиква друга функция, която използва променлива със същото име, освен ако променливите не са били декларирани с *var*, двете двете функции реално ще работят с една променлива и ще променят само нея. Това почти винаги води до неочаквано поведение.

За да включим функцията „option explicit“ е достатъчно да се добави следния коментар в началото на файла, който ще анализираме: */\*jsl:option explicit\*/*. Разбира се това може да се направи и чрез промяна на конфигурационния файл, за да си спестим писането на горния коментар във всеки JavaScript файл, който анализираме.

Ако скрипта използва променливи, функции или обекти от друг JavaScript файл, е необходимо да добавим пътя към този файл чрез коментара */\*jsl:input <path>\*/*. Това ще каже на JavaScript Lint да провери за декларации и в другия скрипт. Относителните адреси трябва да се задават относно пътя на текущия скрипт.

JavaScript Lint може да направи справка и за глобални обекти, които се предоставят по време на изпълнение (например от Mozilla Firefox). Например JavaScript кода в един уебсайт може да се обръща към глобалния обект *window*. За да JavaScript Lint за този обект е достатъчно да добавим реда *+define window* в конфигурационния файл на инструмента. Но JavaScript Lint не валидира свойствата на обекта. Те не използват ключовата дума *var* и не могат да бъдат валидирани без изпълнението на скрипта.

Предупрежденията за необявени идентификатори ще се появят след другите предупреждения, които могат да възникнат в скрипта. Това се дължи на това, че трябва да се разгледа целия скрипт преди идентификаторите да могат да се нарекат неопределени.

Подобно на описаното за JSLint поведение, JavaScript Lint също следи за наличието break оператор след всеки case в switch. Може изрично да се посочи на JavaScript Lint да не връща предупреждение за липсващ break чрез коментара */\*jsl:fallthru\*/*.

По подразбиране JavaScript Lint връща предупреждение при откриване на празно твърдение. Разбира се тази проверка може да бъде избегната. Това става чрез добавянето на */\*jsl:pass\*/*.

**Closure Compiler**

Closure compiler е част от Google Closure Tools. Closure compiler не е типичният инструмент за статичен анализ. Той е инструмент, който прави JavaScript по-бърз за изтегляне и изпълнение. Той оптимизира JavaScript кода. Closure compiler не е компилира JavaScript до машинен код, а до „подобрен“ JavaScript, разбираем за машините, но не и за хората. Closure compiler анализира кода, премахва „мъртвите“ части от него, а останалия свежда до минимум. Освен това проверява синтаксиса, променливите и проверява за обичайни грешки.

Closure compiler може да се използва под три форми: от командния ред, като уеб приложение или RESTful API. От командния ред е наличен като open source Java приложение, на което се задава като параметър списък с файлове, които да бъдат компилирани. Уеб приложението е налично на адрес *closure-compiler.appspot.com/home*. JavaScript кода може да се подаде като URL адрес, на който се намира файла в нашия код, или директно да се въведе на страницата. Същият този адрес приема и POST заявки, чрез които можем да предадем JavaScript кода.

Ползите от Closure compiler са повишената ефективност и анализирането на кода. Closure compiler намалява размера на JavaScript файловете, с което прави зареждането им по-бързо. Освен това връща предупреждения за некоректен JavaScript код и за потенциално опасни операции, помагайки на разработчиците да пишат по-качествен JavaScript с по-малки грешки и по-лесен за поддръжка.

Като цяло Closure compiler има за цел да е поне толкова или дори по-ефикасен от други инструменти за компресиране на код и да подобри максимално времето за изтегляне за уеб приложенията. Също така, това което е по-важно от гледна точка на статичния анализ е, че Closure compiler, за разлика от други инструменти за компилиране на код, може да помогне за откриване на синтактични грешки още по време на разработката (а не при провеждането на динамичните тестове) и откриването на потенциално проблемен код.

В обикновен режим (simple mode), Closure compiler трябва да е в състояние да се справи по-добре от други компилатори, защото използва compiler-like анализ, за да намери допълнителни начини за минимизиране на размера на кода. Например Closure compiler може да вгради директно в кода (inline) някои функции, които се използват само в някои случаи, да преизползва имена на променливи и да изчислява предварително постоянни изрази.

В разширен режим (advanced mode) могат да се добавят специфични пояснения за откриване на труднодостъпни бъгове.

Closure compiler компилира само файлове, които съдържат само JavaScript.

В повечето случаи по-малкия код е по-бърз, тъй като времето за изтегляне на файла обикновено е най-важният фактор за скоростта на уеб приложенията.

Преди да компресира кода, Closure compiler прави анализ, при който се проверя синтаксиса и се търсят често срещани грешки. Подобно на други инструменти за статичен анализ, Closure compiler връща два типа съобщения – грешки (errors) и предупреждения (warnings). Някои типични грешки, за които проверява Closure compiler са опитите за присвояване на числова стойност, която не може да се запише в 32 бита, опит за повторно присвояване на стойност на константа, деление на нула, опит за извършване на математически операции с нечислови стойности (като например символен низ) и т.н. Подобно на JSLint, Closure compiler не приема поставянето на запетая след последния елемент в масив.

Предупрежденията се отнасят до проблеми, които не са фатални за изпълнението на кода, но могат да повлияят върху коректността на работата на програмата. Някои проблеми, за които Closure compiler предупреждава са декларация на невалидна функция, маскирането на променлива от функция със същото име, подаването на функция, който не е конструктор на оператора *new*, използването на недефинирани променливи и др.

В стремежа си да наложи правила, които да гарантират писането на по-качествен JavaScript код, създателят на JSLint Дъглас Крокфорд внедрява в своя инструмент редица ограничения, които разгледахме по-горе. Това е известна гаранция за спазването на добрите практики в езика. Недостатъкът на JSLint е невъзможността за промяна на настройките му. Програмистът е принуден да се съобразява с разбирането на Дъглас Крокфорд за качествен код. В това отношение JSHint е доста по-либерален. Това е и основната му идея. JSHint позволява много по-голяма свобода на разработчика да го настрои по своите предпочитания. Тази му гъвкавост го прави по-добрият избор за анализ на кода. Освен това JSHint е наличен под формата на плъгин за над 10 редактори и среди за разработка, което е още един плюс в негова полза.

JavaScript Lint е също инструмент, който предоставя добри възможности за анализ. Макар да позволява заглушаването на някои предупреждения, JavaScript Lint, подобно на JSLint, не дава на програмиста да променя изискванията относно кода. В това отношение, JSHint си остава по-добрият избор. Все пак не бива да се пренебрегва функцията „option explicit“ на JavaScrip Lint, която може да бъде изключително полезна в някои ситуации.

Трудно е да се сравни Closure compiler с останалите инструменти за анализ на JavaScript код, тъй като анализа не е основната му функция. Логично и акцентът при него не е насочен натам. Closure compiler е по-подходящ за JavaScript код предвиден за уеб приложение, тъй като освен проверката за грешки и лоши практики при писането, компилира кода, правейки го доста по-подходящ за използване в мрежата.

**PyLint**

PyLint е инструмент, който проверява за грешки в Python кода. Той се опитва да наложи стандарт за кодиране и да открива потенциални проблеми. PyLint има много правила, активирани по подразбиране. Инструмента се конфигурира лес, подобно на JSHint, който вече разгледахме. Освен това дава възможност за добавяне на собствени плъгини за проверка.

PyLint показва броя на съобщенията след анализ на кода, както и някои статистически данни за броя на предупрежденията и грешките, открити в различни файлове. Съобщенията са класифицирани в различни категории, като например грешки (errors) и предупреждения (warnings). Ако изпълним PyLint два пъти подред върху един и същ код, той ще покаже статистиката от предишната проверка, заедно с новата. Това позволява по-лесно да видим разликите и дали подобренията по кода са успешни.

И не на последно място, на кода се дава цялостна оценка, в зависимост от броя и тежестта на предупрежденията и грешките. Това е може да бъде особено мотивиращо за програмиста.

Все пак трябва да се има предвид, че съветите на PyLint не бива да се приемат като правила. PyLint може да даде предупреждение дори и за нещо, което умишлено сме направили по даден начин.

PyLint се опитва да докладва колкото се може по-малко грешки, но обикновено не пести предупрежденията си. Това е така, защото се опитва да открие потенциални проблеми, които могат да възникнат в един контекст, но не и в друг. Така PyLint често прави проверка за неща, които не ни интересуват. Като цяло не трябва да очакваме PyLint да не връща абсолютно никакви грешки и не бива да се тревожим, че кодът ни предизвиква много предупреждения.

**PyChecker**

PyChecker е инструмент за откриване на проблеми в Python код. Специфичното за него е, че за разлика от PyLint например, изпълнява кода, за да го анализира. Той засича проблеми, които обикновено се улавят от компилаторите за по-малко динамични езици като C и C++. PyChecker е подобен на PyLint. Поради динамичния характер на Python, някои предупреждения могат да са необосновани, но те са сравнително рядко срещани.

Най-апред PyChecker вмъква отделните модули. Ако възникне грешка, модулът няма да може да бъде обряботен. Това осигурява основна информация за модула. Кодът на всяка функция, клас и метод се проверява за възможни проблеми.

PyChecker следи за наличието на много видове проблеми. Някои от тях са за изпълнение на модул без да е вмъкнат, подаване на неправилен брой параметри на функции или методи, изпълнение на методи и атрибути на класа, които не съществуват, предефиниране на функция, метод или клас в същия обхват като на първата, използване на променлива преди да бъде дефинирани и т.н.

**PyFlakes**

PyFlakes е инструмент за статичен анализ на Python код, подобен на PyLint и PyChecker. Той е насочен към идентифициране на често срещаните грешки по-бързо, без изпълнение на кода.

Основното му предимство пред PyChecker е, че е доста по-бърз. PyFlakes се справя с доста големи проекти само за няколко секунди, за разлика от PyChecker, който губи доста време, за да изпълни кода.

PyFlakes прави проверка за два основни типа проблеми. Първият е за променливи използвани без да са дефинирани или преди да са дефинирани. Вторият е за повторно дефинирани променливи, без да са използвани. Това може да приема различни форми. Например, PyFlakes ще докладва сбъркани имена на променливи, дефиниране на функция със същото име като вече съществуващо, припокриване обхвата на променливи, вмъкване на модул повече от веднъж или на различни модули с еднакви имена и т.н.

Изложените тук инструменти за анализ са най-използваните за Python код. Всеки един от тях има своите предимства и може да бъде използван в зависимост от целите на програмиста. Безспорно най-задълбочен анализ прави PyLint, което го прави предпочитан избор в повечето случаи.

По отношение на начина на работа, PyLint и PyFlakes са идентични. За разлика от тях, PyChecker изпълнява кода, за да го анализира. Това може да е проблем, ако не искаме да изпълняваме кода. Пример за такава ситуация е, когато в кода имаме SQL заявки, чието изпълнение в тестова ситуация не е препоръчително. Освен това, изпълнението на кода прави PyChecker значително по-бавен от другите два инструмента. Друг минус на PyChecker е, че прави проверка само за логически грешки, без да се интересува от стила на кода.

Що се отнася до скоростта на работа, несъмнено лидер е PyFlakes. Сериозен минус на този инструмент, обаче, е липсата на прецизност спрямо PyLint. PyFlakes изпуска голяма част от потенциалните проблеми. Въпреки това е удобен за бърза проверка или повдигане на самочувствието след особено слабите резултати, които PyLint връща.

Съпоставката между трите инструмента дава предимство на PyLint. Подробният анализ, който прави на кода, комбинирано със статистическите данни, които предоставя го правят фаворит.

*Заключение:*

Изборът на инструмент за анализ зависи от много фактори. Важни са освен възможностите на самият инструмент и целите, за които ще бъде използван, нивото на програмиста и други фактори. В различни ситуации за най-подходящи различни инструменти, но това са по-скоро частни случаи. Предвид изложените предимства и недостатъци на разгледаните по-горе пособия, най-подходящи за масова употреба са JSHint и PyLint съответно за JavaScript и Python. Те правят най-задълбочен анализ и дават най-голяма свобода на програмиста без да изискват прекалено задълбочени познания за използването им. Отделно JSHint предоставя удобна и подробна статистика и възможност за сравнение на промените в кода, което е допълнителен плюс.

***3.Инструменти за създаване на документация***

Често софтуерът включва и техническа документация, предназначена за използване от разработчиците. Тя може да бъде във вид на коментари в самия изходен код или обособена в отделни файлове. Предназначението на тази документация е да улесни бъдещата поддръжка и промяна на софтуера.

Добрата документация на кода , независимо дали е за програмисти и тестери , техническа документация за вътрешни потребители или софтуерно ръководство и помощни файлове за крайни потребители , помага на хората , които работят със софтуера да разберат неговите възможности и функции.Добрата документация е специфична, кратка и уместна ,предоставяща цялата важна информация на човека , работещ със софтуера.

**3.1.КАК ДА ПИШЕМ ДОКУМЕНТАЦИЯ:**

1. *Определяне на точната информация, която трябва да се включи*.Документите, в които е включена спецификацията на кода, служат като ръководства за дизайнерите на потребителския интерфейс, програмисти които пишат кода и тестери , които проверяват дали софтуера работи както е предвидено.Точната информация зависи от конкретната програма , но най-често включва някои от следните:

* *Ключови файлове в приложението* .Това може да включва файлове създадени от екипа на разработчиците, достъп до базата данни по време на работа на програмата и други.
* *Функции и подпрограми*.Това включва описание на това какво прави всяка функция и подпрограма , включително и обхвата на стойностите , които приемат входните и изходните променливи.
* *Променливи и константи , и как те се използват в приложението*.
* *Цялостна структура на програмата*.

2. *Определяне каква част от документацията ще бъде в рамките на програмния код и каква част ще бъде отделена от него.*Колкото повече техническа документация е включена в изходния код на програмата, толкова по-лесно ще бъде тя да се актуализира и поддържа едновременно с кода.Минимумът документация , който трябва да бъде в кода е описанието на функции , подпрограми, променливи и константи.

* Ако кодът е особено дълъг, може да бъде документиран под формата на помощен файл, който да бъде индексиран с ключови думи.Това е предимство за приложения, при които логиката на програмата е фрагментирана на много страници и включва редица допълнителни файлове , както и при някои уеб приложения.
* Някои езици за програмиране , като Java и .NET Framework(Visual Basic.NET, C#) имат свои собствени стандарти за документиране на кода.

3. *Избиране на подходящ инструмент за документация*.До известна степен това се определя от езика, на който е написан кода , тъй като съществуват специфични инструменти за всеки език.В други случаи инструментът , който се използва се определя от типът на документацията.

**3.2 JAVADOC*.***

Един от най-полезните инструменти в JDK е JavaDoc . Този инструмент се използва за създаване на документация от коментари , намиращи се в самите Java програми.Тази документация автоматично се преобразува в HTML документ.

**JavaDoc коментари.**

Коментар в JavaDoc започва с /\*\* и завършва с \*/.

Пример за JavaDoc коментар:

*/\*\**

*\* This is a <b>javadoc</b> comment.*

*\*/*

Поради факта , че JavaDoc генерира HTML,всеки валиден HTML таг може да бъде разпознат.В горния пример ‚“JavaDoc“ ще бъде потъмнено.

Коментарът може да бъде на няколко реда:

*/\*\**

*\* This is line one.*

*\* This is line two.*

*\**

*\* This is intended as a new paragraph.*

*\*/*

Независимо, че редовете са разделени в сорс файла, когато инструментът прави разбор на коментарите , символите \* се пренебрегват както и празните символи .Символите \* са за улеснение на този, който чете кода и не се появяват в изходния HTML файл.Може да се използва тага <р> за дефиниране на нов параграф.

Друг полезен таг е <code>, който се ползва за включване на примерен код в коментар.Всеки текст между <code> </code> ще се появи в шрифт Courier.

Първото изречение на коментара трябва да бъде резюме на коментара.Когато инструментът генерира HTML файл, това първо изречение се появява в съдържанието на HTML файла.

За да бъдат разпознати като JavaDoc коментари от инструмента, те трябва да бъдат поместени непосредствено преди класа , интерфейса, конструктора или метода , за когото се отнасят.((Това е и причината поради която важните твърдения се намират в началото на сорс кода , когато се използват JavaDoc коментари.))

**JavaDoc Тагове.**

Има редица специални тагове , които могат да бъдат използвани при JavaDoc коментари.Всички те започват със символа @.Ще споменем най-често използваните .JavaDoc таг трябва да започва в началото на реда.Ако съществуват повече от един ред , започващи с еднакъв таг, тогава обединете таговете.

*@author*

Този таг се ползва , за да се посочи автор на коментара.Може да има няколко @author тага.Той има смисъл само за клас /интерфейс JavaDoc коментар.

*@version*

Използва се за задаване на версия.Може до има няколко @version тага.Версията най-често се отнася до софтуера, който поддържа дадена опция.

*@see*

Този таг се ползва за връзка:“Виж също“ към клас.Пример:

@see java.lang.String

*@param*

Използва се за добавяне на описание на параметър на метод.състои се от две части:първата е име на параметъра, а втората е самото описание, което може да бъде повече от един ред.Пример:

*@param size the length of the passed array*

*@return*

Използва се , за да се добави описание на връщаната стойност.Има смисъл само , ако типът е различен от void.Пример:

*@return true if the array is empty; otherwise return*

False

**Генериране на JavaDoc документация:**

След като добавим JavaDoc коментарите към сорс файла използваме командата javadoc , за да създадем документация.Изпълняваме javadoc по същия начин както и javac или java tools.След името на командата въвеждаме или името на пакета или името на сорс файла.

По подразбиране JavaDoc генерира набор от HTML файлове , един .html документ за всеки сорс файл, който описва public /protected класове , интерфейси, конструктори и методи.В случай , че искаме да променим това , трябва да се въведат опции:

*javadoc -private javabook*

В този пример JavaDoc ще генерира информация за всички класове и методи.

**3.3.VSDOCMAN**

VSDOCMAN позволява коментиране и автоматично генериране на техническа документация от VB.NET и C# сорс код файлове.Той е интегриран във Visual Studio директно като добавка.

VSDOCMAN прави разбор на Visual Studio проекти и автоматично създава съдържание, индекс, API , теми и препратки.

Този инструмент може да чете XML коментари от изходния код.VSDOCMAN съдържа WYSIWYG редактор за коментари, който помага за писането на коментари.Възможно е вмъкването на таблици, списъци, снимки , линкове и форматиране директно върху кода.Възможни формати за изходния код: HTML Help, Microsoft Help viewer , Microsoft Help 2 , HTML, Docx , RTF , XML.

**Предимства.**

* Спестява време , усилия и пари.Документацията се генерира от .NET XML коментари само с един бутон.

A description...

Фиг.3.1:Продуктивност

* Създава професионална техническа документация в множество различни формати – HTML, CHM, Microsoft Help Viewer(VS 2012/2010 help),MS Help 2 (VS 2002 – 2008 help), Docx(OOXML формат използван в MS Word , който може лесно да бъде трансформиран в PDF), RTF, XML и други.

A description...

Фиг.3.2:Изходни формати

* VSDocman може да създава автоматични XML коментари .Шаблоните значително улесняват писането на коментари.

A description...

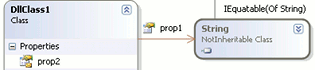
Фиг.3.3:Автоматично коментиране

* Чрез него могат да се добавят таблици , списъци, снимки, линкове и други , директно към XML коментарите.



Фиг.3.4: Редактор на коментарите

* Възможно е добавяне на клас диаграми навсякъде в документацията



Фиг.3.5:Клас диаграми

.

* Създаване на пълно ръководство за ползване на потребителя включващо резюме , примери и други.



Фиг.3.6: Пълно ръководство

**Включва още:**

* Поддържа VB .NET и C# във всякакъв тип проекти, включително и клас библиотеки, приложения , ASP.NET.
* Работи с Visual Studio 2012/ 2010/2008/2005.
* Поддържа всички .NET Frameworks:1.0, 1.1, 2.0, 3.0, 3.5,4.0, 4.5.
* Работи със GUI и режими в команден ред.
* Може да се избере кои части да бъдат документирани, като се използва филтриране по тип , достъп, файл, име или атрибути.
* Могат да се добавят собствени теми като например резюме, примери, лиценз,описание за ползване и други.
* Профили за настройките

**Коментари**

* Поддържа всички стандартни коментари.
* Автоматично вмъква предварително дефиниран текст в коментарите, например „*Initializes a new instance of the <CLASS-NAME-AND-LINK> class“* за конструктори.
* Редактор на коментари , който се използва директно във Visual Studio
* Може автоматично да се добави XML коментар към даден компонент.
* Може автоматично да се добави описание на атрибут.
* XML тагове , които се поддържат : <summary>, <seealso>, <example>, <same>, <typeparam>, <param>, <paramref>, <typeparamref>, <set>, <returns>, <remarks>, <overloads>, <author>, <version>, <revision>, <include>, <includesource>, <compilewhen>, <exception>, <value>, <permission>, <threadsafety>, <para>, <code>, <c>, <see>, <list>, <img>, <b>, <i>, <u>, <br>, <font> .

**3.4.SPHINX**

Sphinx е инструмент , чрез който лесно може да се създаде добра документация.В началото е бил написан да поддържа единствено Python проекти,но в момента може да работи и със C/C++ проекти и се очаква списъкът с поддържаните езици да се допълва.

* *Изходни формати*:HTML( вкл.Windows HTML Help), LaTeX , Texinfo, man pages, чист текст.
* *Връзки*:Семантично маркиране и автоматични линкове за функции, класове цитати и други.
* *Йерархична структура*:Дърво с автоматични линкове към деца, родители и други връзки.
* *Автоматични индекси*:Индекси за всички модули.
* *Код*:Автоматично оцветяване .
* *Разширения*:Автоматично тестване на части от кода.

Sphinx използва reStructuredText като основен маркиращ език и много от неговите силни страни идват именно от това.

**Използване на Sphinx**

* *Дефиниране на структурата на документа*.

При инсталиране на Sphinx се създават два файла – conf.py и главен документ index.rst.Главната функция на index.rst е да служи като заглавна страница и в него се намира съдържанието (toctree).

.. **toctree**::

**:maxdepth:** **2**

intro

tutorial

**...**

Фиг.3.7: Добавяне на компонент към съдържанието

* *Добавяне на съдържание към компонентите.*

В Sphinx сорс файловете могат да се използват повечето функции на стандартния reStructuredText.Но съществуват и много добавки.Например, могат да се добавят връзки между файловете , използвайки ref.

* *Изпълнение.*

$ sphinx-build -b html sourcedir builddir

Фиг.3.8: Пускане на build

sourcedir – директория на сорс кода.

builddir – мястото където ще се съхранява документацията за кода.

$ make html

Фиг.3.9:Създаване на HTML документ

Горния ред генерира HTML документ в директорията , която сме избрали.

Командата make latexpdf извиква инструмента PdfTeX и автоматично генерира pdf документ.

* *Документиране на обекти.*

Едно от главните предимства на Sphinx е лесното създаване на документация за обектите във всеки домейн.Домейн се нарича колекция от типове обекти, които са свързани.

Най-важният домейн е домейнът Python.Например, ако искаме да коментираме вградената функция enumerate() трябва да добавим следния текст към сорс файла:

.. **py:function**:: enumerate(sequence[, start=0])

Return an iterator that yields tuples of an index and an item of the

*\*sequence\**. (And so on.)

Фиг.3.10:Коментиране на вградена функция

Приема се като :

*enumerate(sequence[, start=0])*

*Return an iterator that yields tuples of an index and an item of the sequence. (And so on.)*

Аргументът на директивата е обектът, който се описва , съдържанието е самото описание.

Съществуват още няколко директиви за документиране на обекти : py:class, py:method

* *Автоматично документиране.*

Когато се документира код е нормално да се поставя документация директно в сорс кода.Sphinx поддържа включване на низове с документация с разширение, наречено autodoc.

Пример.За да документираме функцията io.open() трябва да добавим следния ред към сорс файла:

.. **autofunction**:: io.open

Фиг.3.11:Автоматично документиране

Също така могат да се документират цели класове или дори модули автоматично:

.. **automodule**:: io

**:members:**

Фиг.3.12:Добавяне на класове и модули

**3.5.ASCIIDOC**

ASCIIDOC e олекотен маркиращ език, който включва конвертираща програма , която може да преобразува ASCIIDOC документи в XHTML, DocBook или HTML.От друга страна DocBook може да бъде конвертиран в PDF, TeX, Unix manpages и много други , използвайки инструмента A2X , който е включен в конвертиращата програма.Повечето от Git документацията е написана на ASCIIDOC.С помощта на ASCIIDOC могат също така да се пишат бележки, статии , уеб страници, блогове и други.ASCIIDOC е безплатен софтуер и е лицензиран под GNU General Public License Version 2 (GPL v2).

ASCIIDOC документ се пише по същия начин , по който се пише обикновен текстов документ.Няма маркиращи тагове или сложни обозначения.ASCIIDOC файловете са създадени, за да бъдат разглеждани, редактирани, принтирани директно или транслирани към други формати , използвайки asciidoc командата.

Инструментът ASCIIDOC е доста лек:съдържа единствено Python скрипт и набор от конфигурационни файлове.Единствено той и интерпретатора на Python могат да конвертират ASCIIDOC текстови файлове в DocBook или HTML .

**ТИПОВЕ ДОКУМЕНТИ В ASCIIDOC**.

Съществуват три типа ASCIIDOC файлове:статия, книга и manpage.Всички те споделят един и същи ASCIIDOC формат с някои малки изключения.

*asciidoc(1) –d (--doctype)*

Тази команда определя типът на документа – по подразбиране това е статия.Използва се разширението .txt за ASCIIDOC сорс файловете.

* *Статия*.Използва се за кратки документи и главна документация.ASCIIDOC създава стандартна DocBook статия включваща приложение,същност, библиография, речник и индекс.
* *Книга*.Използва се същия формат , както при статия, с няколко изключения:
  + Заглавията на отделните части , са изнесени най-отгоре във вид на съдържание на книгата.
  + Съществуват специфични за този формат компоненти като например предговор.

Документите тип книга се използват обикновено , за създаване на DocBook файлове, тъй като инструментът , който преработва тези файлове може автоматично да създава съдържание,списък от таблици, списък от примери и индекси.

* *Manpage*.Използва се , за да се създават UNIX manpages.Документите от този тип се създават според утвърдени конвенции.

Шаблоните за създаване на секции са дефинирани в конфигурационните файлове.Начини за определяне заглавие на секция (по ред на използване):

* + Заглавието на първия позиционен атрибут или атрибут шаблон.Следните три заглавия на секции са еквивалентни:

[[terms]]

[glossary]

List of Terms

["glossary",id="terms"]

List of Terms

[template="glossary",id="terms"]

List of Terms

* + Когато заглавието съвпада с конфигурационен файл.
  + Ако никое от горните не е изпълнено , по подразбиране се използва sect<level> шаблон , където <level> е число от 1 до 4.

**ASCIIDOC изходни файлове**.

Командата asciidoc(1) превежда ASCIIDOC форматирани файлове в изходни файлове чрез опциите –b(--backend).Изходните файлове могат да бъдат:

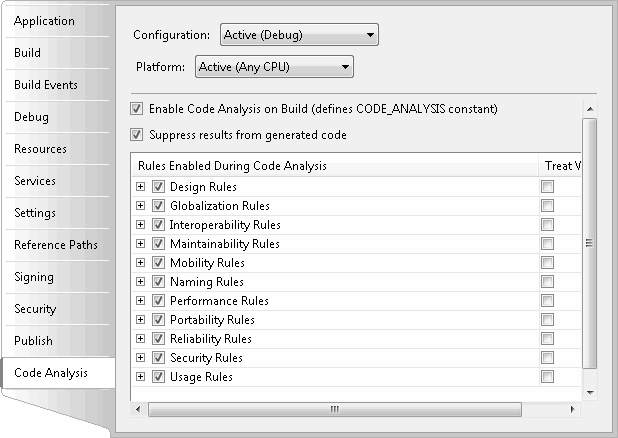
* *docbook45* – създава DocBook XML 4.5 файл.
* *html4* – създава изчистен HTML 4.01 файл.
* *xhtml11* – създава XHTML 1.1 файл , като включва и CSS2.Тези файлове са с разширение .html.
* *html5* – създава HTML5 файл, който е същия като xhtml11, с добавени аудио и видео макроси.
* *slidy* – създава HTML Slideshow файлове от ASCIIDOC документи.
* *wordpress* – вариант на html4 за създаване на постове в блог.
* *latex* – създава latex файлове.Все още е на експериментално ниво.

**Инструменти за анализ на код**

Ще разгледаме употрeбата на инструменти за анализ на кода използвани във Visual Studio, тъй като 90% от операционните системи на стационарни и преносими компютри са продукти на Microsoft, а най – разпространената среда за разработка под тази операционна система е Visual Studio.

FxCop

Първо ще разгледам инструмента за анализ на статичен код – FxCop. Анализа на статичен код без да се изпълнява сорс кода на приложението. За да можем да използваме този инструменти първо трябва да го включим. През properties dialog box на проекта от менюто вляво можем да изберем Code Analysis за да променим настройките на инструмента. Фигура 4.1 илюстрира наличните настройки.



Фигура 4.1: Настройки на проекта за включване на code analysis.

За да включим инструмента трябва да изберем checkbox-a на "Enable Code Analysis on Build...". Тази настройка не изпълнява незабавно анализ на кода. Вместо това индикира, че кода трябва да бъде автоматично прегледан след всяка компилация.

Генериран код

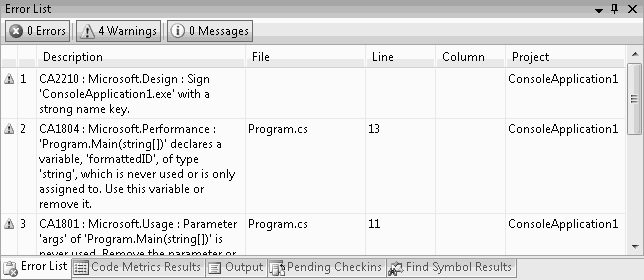
Част от кода на проектите създадени чрез Visual Studiо обикновено е генераран от средата за разработка (IDE). Средата за разработка добавя стандартни елементи като нужните методи за стартиране на приложението и кода който задава настройки при отваряне на Windows form. Този код често включва елементи които не отговарят на изискванията които анализа на кода проверява. За да избегнем проверката на генерирания код можем да включим опцията "Suppress results from generated code" от настройките. Така при компилация, грешките в кода генериран от средата няма бъдат показвани.

Правила за анализ на кода

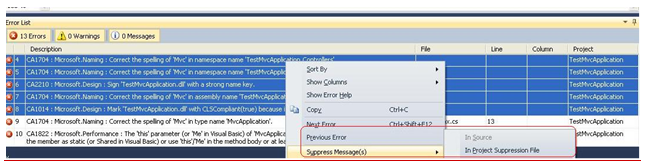
Списък от табове за анализ на кода съдържа списък от правила които се проверяват от процеса за анализ на кода. Тези табове са групирани на разширяващи се секции. Когато едно правило от списъка е нарушено се показва предупреждение. Всяко правило допълнително вдясно има checkbox. Ако той е избран, предупреждението става грешка и спира компилацията да не завърши успешно докато проблема не бъде решен или експлицитно игнориран.

Предупреждения и грешки при анализ на кода

Процеса на анализ на кода се изпълнява когато се компилира проекта. В зависимост от настройките които са избрани в секцията „Анализ на кода“ от настройките на проекта, всяко нарушение на програмните или дизайн правила генерира предупреждение или грешка. Тези грешки и предупреждения се показват в прозореца със списъка с грешки.



Фигура 4.2: изглед от списъка с грешки и предупреждения във Visual Studio.

Всяко нарушение показано в списъка с грешки съдържа подробна информация за нарушението, като име на файла, име на проекта, номер на реда на грешката и т.н. Нарушението може да се поправи или да се настрои така че да бъде игнорирано както е показано на фигура 4.3.

Фигура 4.3: настройка на предупреждение/ грешка да бъде игнорирано/а.

Ще разгледам инструментите, които Microsoft предоставя за покритие на кода. Нека първо изясня какво представлява покритието на кода. **Code coverage** е мярка която се използва в софтуерното тестване. Тя описва до каква степен сорс кода на програмата е бил тестван. За да се измери процента от кода който е бил тестван се изполват един или повече критерии на покритие.

Базови критерии на покритие:

* Functional coverage – Дали всяка функция на програмат е била извикана?
* Statement coverage - Дали всеки възел от програмата е бил изпълнен?
* Decision coverage – Дали всеки edge на програмта е бил изпълнен? На пример, дали изисквания на всяко разклонение на всяка контролна структута (като if и case) са били изпълнени или не?
* Condition coverage – Дали всеки болев подизраз е бил оценен с истина и лъжа..
* Condition/decision coverage -Decision и condition coverage трябва да бъдат удовлетворени.
* State coverage – Дали всяко краино положение на машината е било достигнато и проучено

Пример:

int foo (int x, int y)

{

int z = 0;

if ((x>0) && (y>0)) {

z = x;

}

return z;

}

Допускаме че тази фунцкия е част от по-голяма програма и са били пуснати тестове върху тази програма.

* Ако по време на изпълнението функцията 'foo' е била извикана поне веднъж, тогава *function coverage* за тази функция е изпълнено.
* *Statement coverage за тази функция ще бъде удовлетворено ако е била извикана примерно чрез* foo(1,1) като в този случаи всеки ред от функцията е бил изпълнен.
* Тестове извикващи foo(1,1) и foo(0,1) ще удовлетворят *decision coverage*, като в първия случаи двете условия във if са изпълнени и  z = x; е изпълнено, докато при втория случаи, първото условие (x > 0) не е изпълнено, което прави кода z = x; неизпълнен.
* *Condition coverage* мове да бъде удовлетворено с тестове които извикват foo(1,1), foo(1,0) и foo(0,0). Тези са необходими тъи като в първите 2 случая (x>0)  се оценява като истина докато при третия се оценява като лъжа. В същото време, първия случай дава за (y>0)истина докато втория и третия дават лъжа.

Критични откъм сигурност приложения изискват 100% от някои форми на code coverage. Някои критерии за покритие са свързани. Например Decision coverage е свързано със statement coverage, защото всеки statement е част от разклонение.

Целевия софтуер се изгража със специални опции или библиотеки и/или се пуска в такава специална среда че всяка функция, която тества в програмата се прилага към

При имплементирането на политики за покритие на кода, трябва да се вземат в предвид следните:

* Какви са изискванията за покритие за сертифицане на крайния продукт и какво ниво на покритие се изисква? Типични нива са следните: Statement, Branch/Decision, [Modified Condition/Decision Coverage](https://en.wikipedia.org/wiki/Modified_Condition/Decision_Coverage)(MC/DC), LCSAJ ([Linear Code Sequence and Jump](https://en.wikipedia.org/wiki/Linear_Code_Sequence_and_Jump))
* Покритието на кода ще бъде ли измерено въз основа на тестовете, които верифицират изискванията на системата.
* Може ли обектни код директно да бъде проследен до сорс кода? Определени сертификати изискват покритие на ниво assembly, ако това не е случая тогава се налагат допълнителни верификации върху обектния код за да бъде изпълнена коректността на такъв генериран код.

Тестовите инженери могат да погледнат резултатите от покритието на кода за помощ да изимисля тестови случаи и данни или конфигурационни множества които ще увеличат покритието на кода на вайните функции. Две популярни форми на покритие на кода се изполват от тестващия екип - statement coverage и edge (branch) coverage. Line coverage отчита кои редове от кода са били изпълнени за да завърши теста. Edge coverage отчита кои разклонешени8я са били изпълнени за да завърши теста. И двата метода отчитат метрика на покритие на кода в проценти. Знаяението на тези проценти завси от това какви форми на покритие на кода са били използвани, като 67% branch coverage е по – изчерпателно от 67% statement coverage.

Visual Studio използва block-based statement и condition coverage методи. Block обикновено се дефинира като последователност от инструкции които имат една входна точка и една изходна точка. Ще считаме че една изходна точка може да е разклонена инструкция, извикване на функция, върната инструкция или в случаи на преработен код – хвърлена инструкция.

Пример:

1: int foo(bool condition)

2: {

3: int i = 0; /\* block 0 \*/

4: if (condition) /\* block 0 \*/

5: {

6: i = 5; /\* block 1 \*/

7: }

8: else

9: {

10: i = bar(); /\* block 2 and 3 \*/

11: }

12: return i; /\* block 4 \*/

13: }

Това е генерирания debug x86 на кода отгоре

|  |  |  |  |
| --- | --- | --- | --- |
| Address: | Instruction: | Operands: | Block: |
| 0000 | push | Ebp | 0 |
| 0001 | Mov | ebp,esp | 0 |
| 0002 | push | Ecx | 0 |
| 0003 | Mov | dword ptr [i],0 | 0 |
| 0004 | movzx | eax,byte ptr [condition] | 0 |
| 0005 | Test | eax,eax | 0 |
| 0006 | Je | 0009h | 0 |
| 0007 | Mov | dword ptr [i],5 | 1 (due to branch) |
| 0008 | Jmp | 000Bh | 1 |
| 0009 | Call | Bar | 2 (due to branch) |
| 000A | Mov | dword ptr [i],eax | 3 (due to call) |
| 000B | Mov | eax,dword ptr[i] | 4 (multiple entry points) |
| 000C | Mov | esp,ebp | 4 |
|  |  |  |  |
| 000D | Pop | Ebp | 4 |
| 000E | Ret |  | 4 |
|  |  |  |  |

Забележете че всичко до и включително инструкцията от първия branch (je) се счита за част от първия блок. Всичко до и включително инструкцията (jmp) от втория branch се счита за част от втория блок Третия блок е съставен само от инструкцията за извикване на бар функцията. Четвъртия блок е съставен от запазването на върнатата стойност от извикването на бар функцията в промеливата i. Забележете, че за дефиницията на блок като единствен вход и единствен изход за последователност от инструкции, инструкцията на 0x000B има две входни точки: като следващата инструкция от 0x000A и също от неусловния jump на 0x0008. Следователно, то също се счита за старта на нов блок и всичко до финала като инструкции е от 5 блок. Така както можете да видите, един ред код може всъщност да бъде повече от един блок.

Работата на инструментирането е да модифицира оригиналния изпълним image (dll,exe…), така че да можем да отчетем кои блокове са били “ударени” по време на изпълнение. За да постигнем това, няколко инструкции (в този случай на x86: един push, два mov и един pop) се слагат във всеки блок за да сменят байт в един буфер който казва че този блок е бил изпълнен. Лесно могат да бъдат видяни тези инструкции ако се декомпилира един инструментиран image. Допълнителна информация която се изисква за да можем да съотнесем блок обратно към оригиналния сорс код също се съхранява в инструментирания image . Инструментиращите инструменти разчитат че дебъг информацията ще бъде налична за да изпълнят инструментирането. Тъй като инструментиращия процес модифицира Image-a, нова дебъг база данни от информация ще бъде записана и референцирана от инструментирания изпълним файл. По подразбиране изнструменитарщия инструмент прави на място инструменитрането и презписва image-a, който се инструментира. Оригиналния image се запазва в случай, че нещо се обърка. Инструментираната база данни от дебъг информация се запиива като различен файл, така оригиналната база данни от информация за дебъг не е нужно да се запазва.

Инструментиращия инструмент използван от Visual Studio се нарича vsinstr.exe. Намира се в "%Program Files%\Microsoft Visual Studio 9.0\Team Tools\Performance Tools". За да изполваме vsinstr.exe само трябва да му зададем атрибута /coverage и пътя до Image-а който искаме да инструментираме:

vsinstr.exe /coverage foo.dll

Това ще генерира foo.dll(инструментирано), foo.instr.pdb (инструментирания pdb референциран от инструментирания foo.dll), и foo.dll.orig (запазения оригинал на foo.dll). Веднъж инструментиран image-a е готов за code coverage collection.

Code coverage collection е процес събиране на данни по време на изпълнение от това което тестовите функции са покрили при изпълнението си. Крайния резултат от codecoverage collection ще бъде файл с данни, който може да бъде анализиран за да покаже резултатите от покритието по прегледен за човека начин.

Visual Studio използва отделен инструмент за наблюдение, координация и записване на данните за покритието в coverage файл. Този инструмент е VSPerfMon.exe – наречен "Visual Studio Performance Monitor". Когато започне наблюдението то слуша за идващи връзки от изпълними файлове, които са били инструментирани за coverage. Нещо което не споменах при описанието на инструменитирането е, че когато един изпълним файл е инструментиран, инструментиращия процес съзадава dependency към VSCover90.dll за Image-a които е бил инструментиран. VSCover90.dll съдържа логиката за комуникация с наблюдението. Ако наблюдението не е включено, връзката ще се провали и coverage collection ще бъде невъзможно до края на живота на процеса. Един инструментиран image ще се изпълни нормално без включено наблюдение. Ако обаче наблюдението е включено и опита за връзка успее, VSCover90.dll ще регистрира хостовия процес и ще започне coverage collection.

VSPerfMon.exe може да бъде намерено в "%ProgramFiles%\Microsoft Visual Studio 9.0\Team Tools\Performance Tools". За да стартираме наблюдението трябва да изпълним следната команда от command prompt:

Vsperfmon /coverage /output:<path\_to\_covarage\_file>

Ако искаме да стартираме наблюдението във фонов режим трябва да изпълним следната команда:

Vsperfmon /start:coverage /output:<path\_to\_covarage\_file>

Това ще включи наблюдението в „stand-alone mode“. В този mode можем да видим процесите които са се свързали и после откачили от наблюдението. За да изключим наблюдението трябва да изпълним следната команда:

Vsperfcmd /shutdown

Има няколко неща които могат да попречат на успешния codecoverage collection:

* По подразбиране наблюдението приема само връзки от процеси вървящи по идентификация която е локалния администратор. Най-честия проблем с това е че никакви данни не се събират за ASP.NET сайтове. Това е така защото тези сайтове върват под идентификацията на Network Service по подразбиране. За да се разреши този проблем, има /USER на vsperfmon I vsperfcmd които позволяват специфициран достъп на потребител до наблюдението. Когато изпълняваме code coverage на web саит във Visual Studio, се прави заявка от Visual Studio за да се разбере идентификацията на процеса. После тази идентификация се предава на наблюдението за да позволи връзката. Това се случва автоматично преди изпълнението на тестовете.
* По подразбиране наблюдението създава обекти и комуникационни канали необходими на coverage runtime в областта на локалната сесия. Следователно ако пуснем наблюдението на различна сесия от сесията на процеса който ще се свърже с наблюдението, тогава се налага да специфицираме /CrossSession атрибута към vsperfmon/vsperfcmd. Това ще инструктира наблюдението да създаде тези обекти и канали в глобалната област, така че да бъдат достъпни от други сесии.

VSPerfCmd

В софтуерното инженерство profiling е форма на динамичен програмен анализ който измерва на пример паметта или времето за изпъленение на една програма, употребата на специални инструкции или честотата и продължителността на извикваните функции. Най-честата употреба на profiling информация е с цел оптимизация на програмата.

Profiling се постига чрез инструментиранего на програмния сорс код или чрез инструментиране на двоичната му изпълнима форма като се използва инструмент наречен Profiler. Много различни техники могат да бъдат използвани от profiler-ите, като базиран на събития, статистичен, инструментиран и симулационни методи. Генерирането на отчет от profile – изисква последователност от команди в command prompt – а.Във Visual Studio ролята на profiler играе VSPerfCmd

Дефинираме път до изпълнителните инструменти на Visual Studiо и се местим в папката където се намира приложението ни:

set pt="C:\Program Files\Microsoft Visual Studio 9.0\Team Tools\Performance Tools"

cd [my\_app\_folder]

Най- простия начин за получаване на Profile отчет е следния:

:: Стартиране на profiler

%pt%\vsperfcmd /start:sample /output:my\_sampled\_data.vsp

:: свъзваме приложението чрез profiler-a

%pt%\vsperfcmd /launch:my\_app.exe

::Изключваме profiler-a (тази команда чака докато приложението не завърши)

%pt%\vsperfcmd /shutdown

Нека обобщим всичко дотук като знания за code coverage:

* Инструментирането става чрез vsinstr.exe. За да иснтрументираме image трябва да извикаме следната команда: Vsistr.exe /coverage <path\_to\_image>
* За да включим процеса на наблюдение който ще събере данните за покритието трябва да изпълним следната команда: vsperfcmd /start:coverage <path\_to\_coverage\_file>
* На третата стъпка пускаме тестовете с които искаме да видим каква част от кода сме покрили
* За да спрем collection трябва да изпълним vsperfcmd /shutdown

Примерен скрипт за code coverage:

@**echo** **off**

**set** curdir**=**%~dp0

**set** outputdir**=**.\bin\

**set** xunitfile**=**"xunit.console.clr4.x86.exe"

**set** instr**=**"c:\Program Files\Microsoft Visual Studio 10.0\Team Tools\Performance Tools\VSInstr.exe"

**set** perfcmd**=**"c:\Program Files\Microsoft Visual Studio 10.0\Team Tools\Performance Tools\VSPerfcmd.exe"

mkdir %outputdir%

**del** %outputdir%\*.dll %outputdir%\*.orig %outputdir%\*.pdb %outputdir%\*.exe %outputdir%\*.config %outputdir%\*.coverage

copy ..\XXX.Tests\bin\Debug\\*.dll %outputdir%

copy ..\XXX.Tests\bin\Debug\\*.pdb %outputdir%

copy ..\xUnit\%xunitfile% %outputdir%

copy ..\xUnit\%xunitfile%.config %outputdir%

copy ..\xUnit\xunit.runner.utility.dll %outputdir%

*rem List of code assemblies*

%instr% %outputdir%XXX.dll /COVERAGE

%perfcmd% /start:COVERAGE /OUTPUT:%outputdir%Project.coverage

*rem Run tests (xUnit, NUnit, etc...)*

pushd %curdir%

**cd** %outputdir%

%xunitfile% XXX.Tests.dll

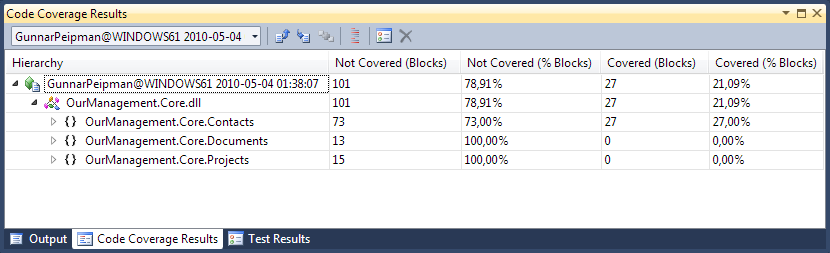
popd %curdir%

%perfcmd% /SHUTDOWN

**call** %outputdir%Project.coverage

@**echo** **on**

Сега имаме coverage файл които можем да използваме за да видим резултата от покритието. Coverage файла може да бъде отворен чрез Visual Studio и данните от него ще се покажат както се показват данните от автоматичен coverage пуснат през Visual Studio:



Фигура 4.4. Пример за резултата от code coverage на проект.

Инструмента VSPerfReport

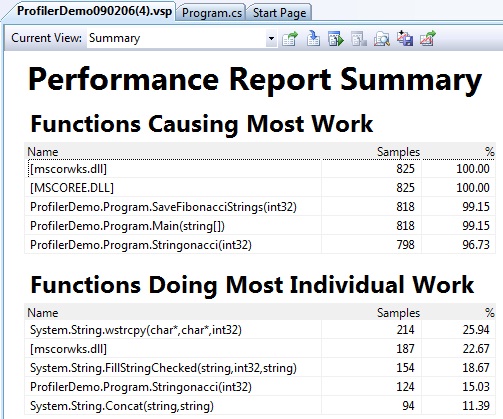
VSPerfReport се използва за създаване на отчети изполвайки данните събрани от profiling. Извиква се използвайки следния синкасис:

VSPerfReport [/U] име\_на\_файла [/опции]

|  |  |
| --- | --- |
| **Опции** | **Описание** |
| *SUMMARY:[types]* | Създава един или повече типове отчети:  *ALL* – всички типове отчет се генерират  *CallerCallee* – отчета съдържа родител/дете връзки между функции  *Function*- отчета съдържа извиканите функции  *Module*- отчета съдържа модулите които са били профилирани.  *CallTree* - отчета съдържа йерархия на извиканите функции.  *Thread* - отчета съдържа профилираните нишки.  *Type* - отчета съдържа алокираните типове.  *Life* - отчета съдържа продължителността на живота на алокираните типове. |
| *CALLTRACE* | Създава списък от влизане и излизане от функции, ETW събития, и отбелязвания |
| *XML* | Отчета се записва в XML format. |
| *U* | Output-а and пренасочен конзолен output на отчета се изписват в Unicode. |
| *SYMBOLPATH:path* | Местоположение на символи |
| *OUTPUT:path|filename* | Местополойение на файа с отчета. |
| *PROCESS:n* | Ограничава данните до определен процес n, н е id на процеса. |
| *THREAD:n [,PROCESS]* | Ограничава данните до определен процес и нишка, n е id на нишката. |

vsperfreport MyProgram.vsp /output:c:\temp /summary:all

И ще получим следното:



4.5.Пример за резултата от употребата на VSPerfReport.

**Софтуерни билд системи и автоматизация на билда**

*Какво е софтуерен билд? Какво представляват билд системите и имат ли те почва у нас?*

*Какво означава автоматизация на билда? Защо се нуждаем от автоматизиран билд процес? Как можем да постигнем това?*

Има много въпроси, на които трябва да си отговорим преди да изясним същността на автоматизираните системи за билд и защо биха ни потрябвали.

На първо място, трябва да си изясним понятието „софтуерен билд“. Когато говорим за компютърен софтуер, терминът с**офтуерен билд** се отнася до процеса на конвертиране на файлове сорс код към самостоятелен софтуерен артефакт (или артефакти), който може да бъде изпълняван на компютър. Една от най-важните стъпки в софтуерния билд е процесът на компилация, където сорс кода се конвертира в изпълним код. Номерът на билда се използва много често като идентификатор на версията на софтуера, където имаме major версия (пр. MyCoolSoftware 6.0) и minor версия, която заедно с major версията служи като уникален идентификатор за конкретния билд (MyCoolSoftware 6.0.3.6777).

Докато за по-прости и малки програми билдът се състои от компилирането на един файл, то при по-сложен софтуер кодът може да се състои от много файлове (дори десетки хиляди), които могат да бъде комбинирани по различни начини, така че да се получат множество различни версии. Ето тук идват на помощ билд системите. Главната роля на една билд система е да превърне четим от човека сорс код в изпълнима програма. Освен това, билд системите поддържат пакетиране на уеб-базирани приложения, генериране на документация и автоматичен анализ на сорс кода, както и много свързани и/или подобни на тези дейности. Въпреки, че точните детайли на този процес варират за различните програмни езици и операционни системи, основните концепции са универсални.

*Тогава какво е* ***билд система*** *всъщност?* Прост отговор няма. При такова разнообразие от езици и среди за разработка, не съществува един-единствен модел, който да успее да представи всички възможни билд системи. Една билд система трябва да може да управлява всякакъв тип активност, включваща транслирането на една форма данни (вход) къмто друга форма данни (изход).

Като начало, във всяка среда за разработка е много вероятно да срещнете някои от следните сценарии, свързани с билда:

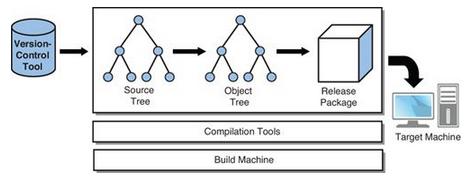
* Компилиране на софтуер, написан на традиционни езици за компилиране, напр. C/C++, Java, C#, Objective-C и др.
* Пакетирането и тестването на софтуер, използващ интерпретирани езици като JavaScript, Python или Perl.
* Компилацията и пакетирането на уеб-базирани приложения. Това включва статични HTML страници, сорс код на Java или C#, хибридни файлове, които използват JSP(JavaServer Pages), ASP(ActiveServer Pages) или PHP(Hypertext Preprocessor), заедно с няколко конфигурационни файла.
* Изпълнението на юнит тестове (Unit tests) за валидация на малки части софтуер изолирано от останалата част от кода.
* Използването на инструменти за статичен анализ за идентификация на бъгове в сорс кода на програмата. Изхода от използването на тази част от билд система е документ, наречен „рапорт на бъгове“, а не изпълнима програма.
* Генерирането на документация в PDF или HTML формат. Това включва използването на различни по тип файлове и генерирането на документация, която е в четим вид.

Това естествено не са единствените функции на една билд система и съществуват много други според конкретните нужди на разработчика. Ще разгледаме малко по-подробно описаните горе сценарии и какво включва всеки един от тях.

**Какви видове билд системи съществуват?**

**Компилирани езици**

При този модел, сорс файловете са компилирани в обектни такива. От своя страна последните се свързват със различни библиотеки и/или изпълними програми. Резултантните файлове биват събрани в пакет, който може да бъде инсталиран на целева машина/устройство. Фигура 1.1 представя традиционната билд система за компилирани езици.



Фиг 5. 1 Схема на билд система при компилирани езици

Ето какво представляват главните компоненти от тази илюстрация:

* **Инструмент за контрол на версиите** (Version-control tool): инструмент, който съхранява сорс кода на програмата и позволява няколко разработчика да правят конкурентни промени в базата код. Също така позволява и достъпването на предишни версии на кода. Често срещани примери за такъв инструмент са CVS, Subversion, Git, Mercurial и ClearCase.
* **Сорс-дървета и обектни дървета**: множеството сорс файлове и компилирани обектни файлове, с който даден разработчик работи. Разработчиците могат да правят свои собствени промени в тези дървета, без да влияят на останалите от екипа.
* **Компилационни инструменти**: приемат входни файлове и компилират изходни такива (например: конвертиране на сорс код в обектен код и изпълними програми). Компилационните инструменти могат да включват и генератори на документация и unit тестове.
* **Билд машини**: машините, на които се изпълняват компилационните инструменти.
* **Пакетиране и целеви машини**: метода, по който софтуера се пакетира, за да бъде доставен до крайния потребител, както и инсталацията му на дадена целева машина.

**Интерпретирани езици**

При интерпретираните езици билд системата значително се различава като модел от тази на компилираните такива. Това може да се види и от илюстрацията долу вляво.

Тъй като интерпретираният сорс код не се компилира до обектен, отпада нуждата от обектно дърво. Вместо това, самите сорс файлове се пакетират, готови да бъдат инсталирани на целева машина. Ако все пак се изискват и компилационни инструменти в този тип билд система (а те често се искат), то те се фокусират върху трансформацията на сорс файловете и съхранението им в пакета за инсталиране. Компилация до машинен код не се прави по време на билд, но може да се случи по време на изпълнение на програмата.

Фиг 5. 2 Схема на билд система при интерпретирани езици

**Уеб-базирани приложения**

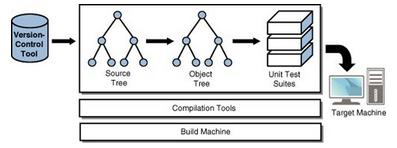
Билд системата за уеб-базирани приложени е микс от компилиран код, интерпретира и конфигурационни файлове. Някой файлове (например HTML файловете) се копират директно от сорс-дървото към инсталационния пакет (release package), докато други първо се компилират в обектен код. Друго специфично при уеб-базираните приложения е, че и уеб сървъра, и браузъра на крайния потребител играят роля в интерпретирането и компилирането на кода. Едно типично уеб-приложение съдържа много от следните типове файлове:

* Статични HTML файлове, съдържащи нищо повече от това, което трябва да се изобрази в браузъра. Тези файлове, както вече споменахме, се копират директно в инсталационния пакет.
* JavaScript файлове, съдържащи код, който трябва да се интерпретира от браузъра на крайния потребител. Могат да се копират директно в инсталационния пакет, но е възможно и да бъдат предварително компилирани; пример за компилатор на JS код е компилаторът на Google, Closure.
* JSP, ASP или PHP страници, съдържащи микс от HTML и програмен код. Тези файлове се компилират и изпълняват от уеб сървър, а не самата билд система. Също се копират направо в инсталационния пакет, готови за инсталация на уеб сървъра.
* Java сорс файлове, които се компилират в обектен код и се пакетират като част от уеб приложението. Билд системата изпълнява тази трансформация преди пакетирането на файловете с Java класовете

Възможно е билд системата да генерира и други HTML/JavaScript/JSP/ASP/PHP файлове посредством други входни файлове. Също така е възможно и още компилационни стъпки да се осъществят преди изходните файлове от целия процес да бъдат добавени към инсталационния пакет.

**Unit Testing**

Билд системата за среда за unit тестване е просто разширение на моделите, описани по-горе. Вместо да създаде инсталационен пакет, билд системата от този тип създава множество по-малки тестови сюити. Всяка сюита се изпълнява на целевата машина и дава резултат – минава ли теста или се проваля. По този начин индикира дали софтуера се държи според очакванията. Следващата илюстрация показва как може билд система за компилирани езици да бъде разширена с модул за unit тестове.

Разбира се, по подобен начин могат да бъдат разширени и билд системите за уеб базирани приложения и за интерпретирани езици. Това показва, че билд системата за unit тестване е просто вариант на коя да е стандартна билд система.

Фиг 5. 3 Схема на билд система с unit тестове

**Статичен Анализ**

При тях се използва инструмент за статичен анализ на кода, като Coverity Prevent, Klocwork Insight и FindBugs. Инструментът проверява сорс кода с цел намиране на потенциални бъгове. Анализа се прави статично (тоест по време на билд), а не чрез изпълнение на софтуера и проверяване дали се държи коректно. Входът на система за статичен анализ е същия сорс код, който се използва при обикновена билд система. Изхода на тази система е документ с рапорт за намерените дефекти.

**Генериране на документация**

Примерния изход от тази билд система би могъл да бъде PDF файл, колекция от HTML страници и графични изображения, или каквото и да е друго, което би могло да се счита за документация. Генерирането на документация може да включва набор от междинни файлове с данни, така че концепцията за обектно дърво е приложима тук.

*Като обобщение до тук, можем да кажем, че една билд система може да се използва за различни и разнообразни цели, но съществуват и множество общи концепции и компоненти между различните видове билд системи.*

**Какво съдържа една билд система**?

**Инструменти за контрол на версията** - това е първия компонент на билд, тъй като преди какъвто и код да бъде компилиран, разработчиците трябва да получат свое собствено копие на сорс кода. Като част от работата си (оправяне на бъг или добавяне на нова функционалност), всеки програмист трябва да промени подходящите файлове и тогава да използва билд системата, за да компилира софтуера. Инструментите за контрол на версията позволяват следните операции:

* Да получите копие на сорс кода, за да можете да правите собствени, вътрешни промени, без да се отразяват на останалите от екипа, докато не “commit”-нете промените си.
* Да внасяте промени в кода (commit), така че другите да могат да ги видят, а също така и да обновят версиите си.
* Да поддържа създаването на множество версии на един продукт
* Да контролира достъпа до файлове, така че само оторизирани разработчици да могат да променят определени файлове.
* Да позволи на програмиста да види предишни, по-стари версии на кода, а също така и да сравнява промените.

**Сорс- и обектни дървета** – подреждането на файловете със сорс код на дадена програма в различни директории е познато като сорс-дърво. Начина, по който сорс кода е структуриран в сорс-дървото има значително влияние на дизайна на билд системата. Структурата на сорс-дървото често отразява архитектурата на софтуера.

При обектните дървета въпреки, че е възможно обектните файлове да бъдат съхранявани в същите директории като сорс файловете, това се приема като объркан подход. Честа практика е да се създаде отделна йерархия, съхраняваща обектните и изпълнимите файлове, получени от билд процеса. Има няколко основни причини, поради които бихме искали да разделим програмата на множество сорс файлове в различни директории.

* *Разбиране*: за хората е по-лесно да мислят за програмите, когато са разделени на логически подсекции. Това е и основна концепция в обектно-ориентираното програмиране, при което хората мислят за програмата като колекция от различни класове с различна функционалност. Всеки клас има външно поведение и вътрешна имплементация, която скрива сложността на класа. Затова в билд системата е най-добре да разделим сорс кода на множество секции, всяка от които капсулира специфична арея от програмната функционалност.
* *Контрол на сорс кода*: Когато сорс кода на една програма е разпръснат между множество файлове и директории, става много по0есно да ги управляваме с инструмент за контрол на сорс код. Противно, ако цялата програма беше 1 файл, щеше да е трудно различни разработчици да качват промени в кода, без да си пречат постоянно.
* *Изпълнение (performance)*: редакторите и компилаторите се справят много по-лесно с по-малки единици. Въпреки, че тези инструменти могат да се справят със сорс файлове, които са мегабайти по размер, те го правят неефективно.

**Инструменти за компилация и билд**

Когато работим със сорс-дърво, ни трябва начин да транслираме четимите от човека сорс файлове във четима от машината изпълнима програма. *Компилационният инструмент* е програма, която чете входни файлове и генерира изходни такива – това може и да звучи като изключително общо твърдение, но всъщност се обуславя от това, че няма ограничение на типа данни, транслирани от компилатора. Примери за компилационни инструменти:

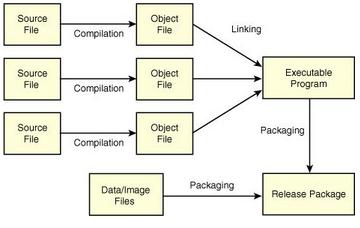
* C++ компилатор - чете C++ сорс файлове и създава обектни файлове, който съдържат машинен код на същата програма .
* Свързващ редактор (*linker*) – свързва различни обектни файлове, за да създаде единствено изпълнимо изображение на програмата. В този случай, обектните файлове са вход на свързващия редактор, докато при предишната стъпка в билда са били изход на компилатора.
* UML-базиран генератори на код – чете файл с UML модел като вход и възпроизвежда еквивалентна програма написана на език за общи цели като Java, C++ или C#.
* Команден инструмент за създаване на нова директория – създава нова директория във файловата система(примерно mkdir в UNIX). Името на директорията е единствения вход.
* Оттук можем да заключим, че компилаторът е инструмент, който приема като входни данни – сорс-файлове, а компилационният инструмент е обобщаващо понятие за инструментите, които приемат като вход едни данни и ги транслират в друг тип изходни такива.

**Билд машини**  - билд машина наричаме машината, на която се изпълняват компилационните и билд инструментите. Тя играе жизнено важна роля в управлението на билд системата. Всеки от инструментите трябва да може да се изпълнява на машината, дори и хардуера или операционната система да се променя отвреме-навреме.

Трябва да се отчете дали самият софтуер се компилира и изпълнява на един и същ тип машина или напротив, софтуерът трябва да се използва в съвсем различна среда(подлежащ хардуер и ОС). Така се създават два вида среди – на естествена (native) компилация и на крос компилация. В случая при естествената компилация, целевата машина, на която се изпълнява софтуера е идентична с билд машината. При крос компилацията двете машини не са идентични.

**Пакетиране** – въпреки, че голяма част от работата на един билд инструмент е да генерира обектни файлове и изпълними програми, крайната стъпка на пакетиране е тази, която произвежда нещо, което наистина може да се инсталира на потребителска машина.Финалната стъпка на билд процеса е да се вземат точните файлове от сорс- и обектните дървета и да се съхранят в инсталационен пакет.

**Какво представлява билд процеса?**



Фиг 5. 4 Илюстрация на билд процес

Това е визуализация на билд процес, която е лесно разбираема от хората, но един билд инструмент се нуждае от описание на билда. То трябва да е написано в текстов формат. Например, когато използваме make-базиран инструмент, вътрешната зависимост между файловете е посочена под формата на правила, които се съхраняват в файл, наречен makefile. При SCons, аналога на make-базираните билд инструменти, се използват функции на питон, за да опишат компилационните стъпки. Ето какво би било описанието на билда при SCons за една малка програма, наречена stock.

Program(“stock”, [ticker.c”, “currency.c”])

В този случай, SCons използва C компилатора по подразбиране, за да създаде обектните файлове ticker.o и currency.o, въпреки, че описанието не посочва изрично тази стъпка. След това обектните файлове се свързват в крайната изпълнима програма stock.

**Какви различни задачи изпълнява билд системата?**

Има три основни типа билдове, които се изпълняват често:

* **Private, developer build** – програмиста е свалил сорс кода чрез инструмента за контрол на версиите и прави билд на софтуера на вътрешна работна станция. Резултатът от това е инсталационен пакет, който програмиста използва за собствени цели.
* **Release build** – един или повече инженери отговарят за тях. Единствената цел на този тип билдове е да осигури пълен софтуерен пакет, които тестерите трябва да валидират. Когато тестерите са убедени, че софтуера покрива необходимите критерии за качество, същия пакет става достъпен и за клиентите
* **Sanity build** – подобен е на release build-а, с изключение на това, че не е предназначен за клиентите. Вместо това билд процеса определя дали сорс кода във системата за контрол на версиите е адекватна, т.е. дали билдът няма грешки и минава основни тестове за адекватност. Този тип билд се случва много пъти на ден и е напълно автоматизиран. Много програмисти наричат този тип билд дневен или нощен билд.

**Как можем да управляваме билд системата?**

Съществуват инструменти за управление на билдовете, те не са точно част от системата (в смисъла на компонент), а по-скоро са допълнителен слой за управление на върха на съществуваща такава. Инструментът за управление си комуникира с системата за контрол на версиите, за да получи билд дърво. Компилира софтуера и дава известие на програмиста дали билда е завършил. Как да разпознаем добрия инструмент за управление на билда? Той трябва да предлага следните възможности:

* Изтегля и прави билд на сорс дърво спрямо предварително определен график, или просто когато нов код е бил добавен.
* Осигурява опашков механизъм, така че множество билд задачи могат да споделят множество (pool) от билд машини. Когато има достатъчен брой налични машини, се стартира следващата задача.
* Изпраща известия (най-често e-mail) до определени групи потребители, например когато почне, завърши, успее или се провали билдът.
* Съхранява финалния софтуерен пакет в архивна директория, готов за тестерите.
* Изпълнява „sanity” тестове (тестове за адекватност) на всеки успешен билд.
* Може да идентифицира кои програмисти са в списъка с „виновници“, когато билдът не успее.

**Какво определя дали една билд система е качествена?**

Както и при всичко останало, няколко неща определят дали една билд система се възприема като „добра“. Можем да я квалифицираме като такава, ако притежава следните характеристики:

* **Удобство**: работи се лесно с нея, дескриптивните файлове са лесни за употреба, така че да не са голям „товар“ за програмиста. Той трябва да се фокусира на писането на код, а не да се справя със сложността на билд системата.
* **Коректност**: билд системата трябва винаги да компилира/свързва правилните файлове, използвайки правилните опции на компилатора. А когато редът има значение, трябва да компилира файловете в правилния ред, така че крайната програма винаги да отразява съдържанието на сорс файловете.
* **Изпълнение**: в един идеален свят (какъвто програмисткия определено не е), билд процесът завършва без забележимо закъснение. В реалността обаче, трябва да бъде изпълнен възможно най-бързо за средата, в която се изпълнява.
* **Мащабируемост**: Трябва да бъде удобен, да осигурява правилни крайни пакети и да се държи добре, дори когато трябва да се направи билд за голяма програма.

**Защо ни трябва автоматизирана билд система?**

Билдът е мониторът на сърцето на вашия проект. Защо? Той ви дава важни предупреждения дали вашия проект е „здрав“. Чрез билда установявате потенциални проблеми със софтуера и дори какво може да ги е причинило. Затова честия, регулярен билд е жизненоважен за вашия проект. Ето защо една добра билд система би могла да ви спести време за в бъдеще. Най-добрите билд системи не само извършват билд, но също така и редица тестове на софтуера, даващи критична обратна информация. Тя помага освен за подобряването на качествата на софтуера, така и за цялостната продуктивност.

Къде е ролята на автоматизацията в случая? Скриптираните билдове позволяват билд само с изпълнението на една единствена команда пот командния ред. Инструменти като Make, GNU Build System и Ant са често използвани, за да се посригне това. Плюсовете от това са, че те позволяват големи части от билда да бъдат скриптирани декларативно и поддържат функционалност като анализ на зависимостите.

**Какви инструменти за автоматизация съществуват?**

***Какво представлява Make?***

*Make* е създаден през далечната 1977 и е първият софтуерен инструмент за автоматичен билд на изпълними програми и библиотеки от сорс код, като чете файлове, наречени make-файлове (makefiles). Тези файлове определят как да се получи целевата програма. Въпреки, че разнообразните среди за разработка и езиково-специфичните опции на компилаторите могат да се използват за управление на билд процеса, Make остава широко използван, особено в Unix средите.

Съществуват много модерни версии. Make е бил преписван множество пъти, включително и няколко пъти изначално, използвайки същите файлови формати и основни алгоритмични принципи, а също и предлагащи свои собствени нестандартни разширения. Популярни варианти на Make са:

* *BSD Make* (*pmake*), който е продукт на работата на Адам де Боор върху версия на *Make*, която да е способна да прави паралелни билдове. Особеното при тази версия е, че има условни и итеративни цикли, които се прилагат още по време на синтактичния анализ и могат да се използват, за условно и програмно построяване на make-файла, включително и създаване на цели по време на изпълнение.
* *GNU Make* е често използван в комбинация с *GNU Build System*. Неговите отклонения от традиционния *Make* са най-забележими при съпоставянето на образци при графовете на зависимости, както и функциите, които могат да бъдат извикани, позволявайки функционалност като показването на списък с файлове в текущата директория. *GNU Make* е също включен и в средата за разработка на Apple, *Xcode*.
* *Makepp* е отмяна за *GNU Make*, а неговите фийчъри са: голямо подобрение при справянето с билдове, съдържащи множество make-файлове (отпада нуждата от рекурсивен make), автоматично сканиране за включващи файлове, стартира се ребилд, ако се промени някоя билд команда, методи за проверяване на билдовете чрез чексума, без да включва whitespace промени или промени в коментарите; позволява разширяване чрез Perl (в самия make-файл)；хранилища, т.е автоматично включва файловете от друго дърво; билд кешове – не компилира наново, нещо което потребител със същия кеш е направил вече.
* Microsoft *nmake* – използва се в Windows среда и е относително прост, от гледна точка на това, че предлага едва подмножество на гореспоменатите версии. Не е същия инструмент като *nmake* на AT&T Corporation Bell и Labs за Unix.

**Поведение на работа на *Make***

*Make* обикновено се използваза билд на изпълними програми и библиотеки. Въпреки това *Make* е приложим към всеки процес, който включва изпълняването на избрани команди с цел трансформация на сорс файлове към целеви резултат. Например, *Make* може да се използва за да открие промени направени към сорса и тогава трансформацията трябва да конвертира файла в специфичен формат, копира резултата към система за управление на съдържанието, след което праща e-mail на предопределена група потребители, известявайки ги, че горните действия са се случили.

*Make* се извиква със списък с имена на файлове за билд като командни аргументи.

make TARGET [TARGET ...]

Без аргументи, *Make* извършва билд на първата цел, която се среща в неговия make-файл.

Решава дали една цел трябва да бъде регенерирана като сравнява времената на модификация на файловете. Това решава проблема с билда на файлове, които са непроменени, но се проваля, когато файлът се промени, а времето му на модификация си остава в миналото. Такива промени могат да бъдат причинени от възстановяването на по-стара версия на файл или когато мрежова файлова система е източника на файлове и нейния часовник или часова зона не са синхронизирани с тези на машината, на която върви *Make.* Потребителят може да избегне тази ситуация като изрично пусне пълен билд. В обратния случай, когато по някаква причина времето на модификация на някой файл се окаже в бъдещето, се стартира ненужен билд.

**Makefiles**

*Make* търси текущата директория за *makе-файла,* който трябва да използва, напр. *GNU Make* търси сред файловете поред за файл, който се казва GNUmakefile, makefile или Makefile и след това изпълнява определените (или тези по подразбиране) цели единствено от този файл.

Езикът, използван за makе-файловете е подобен на езиците за декларативно програмиране. При него необходимите крайни условия са описани, но реда, в който действията се изпълняват не е важен и това е понякога доста объркващо за програмистите свикнали със императивното програмиране.

Един от проблемите при автоматизираните билдове е приспособяването на билд процеса за дадена платформа. Например компилаторът за дадена платформа много често не приема същите опции като компилатор за друга платформа. Това е и слабо място на *Make*. Този проблем обикновено се решава чрез генерирането на платформено-специфични билд инструкции, които на свой ред се обработват от *Make.* Популярни инструменти за този процес са *Autoconf* и *CMake,* безплатни мултиплатформени инструменти за автоматизиране на билд процеса.

***GNU Autoconf***използва конфигурационен скрипт, подготвящ софтуерния пакет за инсталиране на определена целева система. След изпълнението на серия от тестове на целевата система, конфигурационния скрипт генерира заглавни(header) файлове и makefile от шаблони, като по този начин наглася софтуерния пакет за целевата система. Заедно с *Automake* и *Libtool* формира *GNU Build System*. Autoconf е критикуван за това, че използва остарели технологии, че има много останали от миналото ограничения, и по тези причини понякога съвсем простички сценарии биват усложнени ненужно. В частност, често цитирани слаби места на Autoconf са:

* Голяма обща сложност на използваната архитектура, повечето проекти използват множество повторения;
* Конфигурационния скрипт е написан на Bourne shell и оттам генерирането на makefile-а става бавно.
* Слабата съвместимост с предишните и следващите версии изисква „обвиващ“ скрипт
* Генерираните от *Аutoconf* скриптове обикновено са дълги и доста сложни. Въпреки, че предлагат обширен лог, дебъгването на тези скриптове е доста трудно.

Поради тези причини някои разработчици предпочитат да заменят Autoconf (и в частност GNU Build System) с по-лесни за употреба алтернативни варианти като *SCons* и *CMake*.

***CMake,***както вече споменахме е алтернатива на *Autoconf*, проектирана така, че да поддържа множество директорийни йерархии и приложения, които зависят на множество библиотеки, а също така и за употреба с „естествени“ билд среди (т.е не използващи емулатор или виртуална машина) като Xcode и Microsoft Visual Studio. Има минимален брой зависимости - изисква единствено C++ компилатор.

Справя се както с билдове „на място“, така и с „не-на място“, позволявайки няколко билда от едно и също изходно дърво, както и крос-компилация. Възможността да се построи директорийно дърво извън сорс дървото е важен фийчър, подсигуряващ това, че дори директорията на билда да бъде премахната, сорс дървото остава неафектирано.

Освен това, CMake поддържа и се справя добре с сложни йерархии и с приложения, които зависят от няколко библиотеки. Може да работи с проекти, които изискват изпълними програми да бъдат създадени преди генерирания код да бъде компилиран за крайното приложения. Друг плюс на CMake е, че може лесно да бъде разширен според нуждите на специфичния проект. Генерира makefile за Unix, Windows, Mac OS X, OS/2.

**Пълна автоматизация на билда и Continuous Integration (CI)**

Превръщането на сорса в работеща система както вече установихме е сложен процес, включващ компилация, преместване на файлове, зареждане на схеми и т.н. Но както повечето неща дори стъпките преди и след самия билд могат да бъдат автоматизирани – и следователно трябва да бъдат автоматизирани. Колкото по-малко човешка намеса, толкова по-малък шанс за грешки.

Често срещана грешка е да не се включи всичко в автоматизацията на билда. Какво имаме предвид – ако вземете току-що преинсталирана машина, изтеглите сорс кода от хранилището и изпълните една-единствена команда, то трябва да имате работеща система накрая само с извършването на тези действия.

След като вече имате пълен автоматизиран билд, въпроса е колко често трябва да бъде извършван? Отговорът е: възможно най-често. Един отличен начин да направите график за автоматични билдове е всеки път, когато някой направи промяна в кода (не в собственото си копие, а в хранилището), да се стартира билд.

Разбира се, големите билдове често отнемат време, а когато се нуждаете от чести, регулярни билдове, не искате да правите пълен ребилд и да изпълнявате всички описани до сега стъпки всеки път, когато направите и най-малката промяна в кода. Затова добрият билд инструмент анализира всичко, което трябва да бъде променено. Най-лесно прави това като сверява датите на сорс файловете и на обектните файлове и компилира само ако датата на промяна на сорс файла е по-късна от тази на обектния файл. Зависимостите обаче са проблемни – ако един обектен файл се промени, то и тези, които зависят от него, също трябва да преминат билд.

Чрез непълен билд се решават и проблемите на дългите билдове, където отнема време докато се открият проблемите и получим обратна връзка.

В заключение – добре проектираната билд система е един от най-добрите приятели на програмиста. Без съмнение автоматизираните, регулярни билдове биха спестили много време, ценни ресурси и енергия, които биха могли да бъдат вложени в самия проект. Билд системата ви работи за вас – тя не само се грижи за компилацията и свързването на файловете в единна програма, но и ви дава обратна връзка и е монитор – индикира потенциални проблеми, а дори и ги изолира – което прави проследяването на нововъзникнал проблем далеч по-лесно. Затова не я пренебрегвайте и тя ще ви служи вярно.

***Библиография:***

* http://en.wikipedia.org/wiki/JSLint
* http://www.jslint.com/lint.html
* http://jshint.github.io/site/
* http://www.jshint.com/docs/
* http://www.jshint.com/install/
* https://bitbucket.org/logilab/pylint
* http://stackoverflow.com/questions/1428872/pylint-pychecker-or-pyflakes
* http://divmod.readthedocs.org/en/latest/products/pyflakes.html
* http://python.dzone.com/articles/pyflakes-passive-checker
* https://launchpad.net/pyflakes
* http://docs.pylint.org/index.html
* http://pychecker.sourceforge.net/
* http://en.wikipedia.org/wiki/Google\_Closure\_Tools
* https://developers.google.com/closure/compiler/
* https://developers.google.com/closure/compiler/docs/error-ref
* https://developers.google.com/closure/compiler/faq
* http://www.javascriptlint.com/docs/index.htm
* Article 4: Documenting, Commenting, and Documentation Generators - <http://endlesstrain.com/2012/04/15/article-4-documenting-commenting-and-documentation-generators/>
* How to write software documentation - <http://www.wikihow.com/Write-Software-Documentation>
* Javadoc tool, Wen-Yen Chen - <http://alumni.cs.ucsb.edu/~wychen/cs5ja/doc/javadoc.pdf>
* Javadoc, Ashley J.S Mills –
* <https://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/javadoc/javadoc.pdf>
* Special Topics N.1 – Javadoc Documentation Tool –
* <http://www.drcaffeine.com/additionaltopics/St1.pdf>
* 6.VSdocman - <http://visualstudiogallery.msdn.microsoft.com/C6649C4D-38F8-4626-BE3F-E6FBC8B9F679>
* VSdocman - <http://www.helixoft.com/vsdocman/features.html>
* Sphinx – Python documentation generator - <http://sphinx-doc.org/>
* AsciiDoc cheatsheet - <http://powerman.name/doc/asciidoc>
* AsciiDoc - <http://www.methods.co.nz/asciidoc/>
* <http://www.codeproject.com/Articles/144643/Profiling-of-C-Applications-in-Visual-Studio-for-F>
* <http://en.wikipedia.org/wiki/Profiling_(computer_programming)>
* <http://blogs.msdn.com/b/phuene/archive/2007/05/03/code-coverage-instrumentation.aspx>
* <http://msdn.microsoft.com/en-us/library/ms182403.aspx>

* <http://msdn.microsoft.com/en-us/library/ms182402.aspx>
* <http://msdn.microsoft.com/en-us/library/ms182404.aspx>
* <http://en.wikipedia.org/wiki/Build_automation>
* <http://en.wikipedia.org/wiki/CMake>
* [http://en.wikipedia.org/wiki/Make\_%28software%29](http://en.wikipedia.org/wiki/Make_(software))
* Peter Smith, “Software Build Systems: Principles and Experience”, 2011,

ISBN: 9780132171939

* Coding Horror: The Build Server – Your Project’s heart monitor -<http://codinghorror.com/blog/2006/10/the-build-server-your-projects-heart-monitor.html>
* Jason Stankey, “The Road to Build Enlightment” - <http://zutubi.com/products/pulse/articles/buildenlightenment/>
* Martin Fowler, Continuous Integration -<http://www.martinfowler.com/articles/continuousIntegration.html>