Лекция 3 (18. Октомври, 2011)

Sites
SVN интегриран в Explorer: http://tortoisesvn.tigris.org/
Repository на тетриса: svn://anrieff.no-ip.org/tetris

Първи час
Понякога разработването на по-голям проект (над 500 реда) налага изисквания за поддръжка на резервни копия (backup-и) и/или запазване на по-стари версии. Нека например пишем задача за конкурс по програмиране (от типа на конкурса на Мусала софт и PC Magazine). Тези задачи обикновено се дават за период от един месец, като крайният срок е строго фиксиран (12 часа вечерта). Тъй като решенията се предават на автоматизирана система, закъсняване с дори 1 минута след крайния срок не се толерира. Нека сме написали някакво решение, което сме тествали и работи сравнително добре (обикновено в този тип конкурси няма перфектни решения). Няколко часа преди крайния срок се сещаме хубава оптимизация, която би ни донесла няколко допълнителни точки, а защо пък да не дори победата в съответния кръг. Ако просто седнем да я имплементираме има голям шанс времето да не ни стигне и не просто да не предадем по-добро решение – ами да сме „счупили" работещото такова, така че да не предадем НИКАКВО решение! (trivia: това ми се е случвало) Вместо това, можем преди да почнем да пишем рискованата оптимизация, да копираме проекта някъде и да работим върху копието. Това, както най-вероятно знаете, се нарича "backup". Поддържането на бекъпи е много полезно, но изисква време и място на харддиска. Трябва да изберем точните моменти, в които проектът ни е в хубави състояния за да правим бекъпи, трябва да измислим начин да организираме бекъпите така, че да можем да се ориентираме лесно в кой какво е написано и какво не е. Едно не-лошо решение е да запазваме различните копия в различни директории с дати (или някаква номерация на версиите). Във всяка директория да имаме по един текстов файл, който съдържа информация за това какво точно е написано, какво е в процес на писане и какво се планира да бъде написано. Недостатъците на този подход са, че всеки бекъп отнема време за създаване и потенциално много място на харддиска. Представете си, че имате проект, който е 100 мегабайта – само 10 копия биха довели до 1 гигабайт заето пространство!
Макар и горният подход да е относително приемлив за разработка на проект от един човек, какво става, когато разработчиците станат двама? Те трябва да обменят през цялото време своите промени. А често се случва промените на единият да нарушават работата на промените на другия и обратно. Освен това проектът трябва да се „премята" между двете работни станции на двамата човека при всяка по-съществена промяна. Ако след време се открие бъг понякога е доста трудно да се определи в следствие на чии промени е възникнал той, и какво трябва да се смени за да се оправи.
А какво става когато хората станат трима или повече? Каша. В следствие на това са възникнали така наречените системи за контрол на кода. Това са програми, които решават (или поне намалят значително работата за избягване на) голяма част от възникващите проблеми при разработка на софтуер от много програмисти. Един от най-разпространените (предимно заради леснотата за

работа с него) такива клиенти е Subversion (SVN). Именно него и ще разгледаме този час, а и в остатъка от курса.
Идеята на version control системите е следната: на няколко (потенциално един) сървъра се пази целият проект, който се разработва от множеството програмисти. Мястото (URL-то), където се пази проектът, се нарича "repository" (trivia: на някои места в България съм чувал да го наричат „хранилище"). След всяка промяна (била тя прост бъг-фикс от няколко реда или промяна на огромна логика върху стотици или хиляди редове), даден програмист уведомява SVN за промените, които е направил и те се запазват като нова версия на централните сървъри. Забележете, че проектът не се копира изцяло, а се запазват единствено промените! Това значително намалява изразходваното дисково пространство дори при хиляди версии на софтуера. При създаването на тази нова версия (която се нарича „revision") програмистът, който прави промените, обикновено оставя и информация какво точно е променено (примерно „замених линейното търсене на даден user с хеш-таблица с цел оптимизация на query-тата"). SVN, от своя страна, запазва датата и часа на промяната, кои файлове са били променени и какво точно е променено. Забележете, че тъй като се пазят промените, те лесно могат да бъдат премахнати, като така получаваме не какво да е а именно старата версия! Така можем да достъпим ВСЯКА версия, която някога е била създадена.
Освен, че програмистите могат да променят сорса, голям бонус е, че могат и да го теглят оттам. Така всичко, което трябва да се даде на един нов програмист за да започне да работи по кода е едно просто URL, (repository-то, където се съхранява кодът).
Въпреки, че съществуването на такава система значително опростява съвместната работа по един проект, някои от проблемите не се елиминират напълно. Например все още е възможно двама програмиста да правят промени по един и същ сорс, като промените им са несъвместими. Това е немалък проблем (всъщност е един от най-сериозните). Затова функционалността на SVN освен за съхранение на сорс кода (всъщност за всякакви неща – примерно картинки, екзета, библиотеки, каквото ви дойде на ум) е разширена значително. Ето и основните функции на SVN:

1. Checkout
2. Commit (Checkin)
3. Add
4. Delete
5. Show Log
6. Update
7. Update to revision
8. Revert
9. Get Lock
10. Release Lock
11. Branch
12. Merge
13. Create Patch
14. Apply Patch
15. Import
16. Export

17. Info
18. Stat

Сега ще ги разгледаме с по няколко думи коя какво прави и за какво е полезна.

1. *Checkout* – изтегляне на целия проект. Ако сме нов програмист и сме първи ден на работа, най-вероятно това ще ни се наложи да направим, за да имаме проекта при себе си. Тегли се последната налична версия (revision).
2. *Commit (Checkin)* – качват се на централния сървър промените, направени локално. Полезно като сме свършили с някаква промяна на кода (примерно бъг-фикс) и искаме тази промяна да достигне до останалите във фирмата.
3. *Add* – добавяне на нов файл (trivia: директорията също е файл) към проекта. Логично, ако сме създали нещо ново и то трябва да е в проекта, това е начина да се добави в SVN. Забележете, че ако просто създадем файла в работната директория и дадем commit, той НЯМА да се добави в repository-то ако не сме го добавили с командата add!
4. *Delete* – обратното на Add – ако искаме да премахнем някой вече ненужен файл от проекта. Отново забележете – ако просто го изтрием от работната ни директория и дадем commit, това няма да доведе до изтриването му от централния сървър и при следващия update ще се появи отново при нас. След Add и Delete трябва да изпълните commit за да се изпълнят командите в repository-то.
5. *Show log* – показва лог на промените по ревизия. Това са именно коментарите, които програмистите оставят след всеки commit. Това е много полезно за намиране на някоя интересуваща ни версия. Примерно последната, в която не е имплементирано нещо, или първата, в която е добавено нещо ново.
6. *Update* – проверява дали проектът в работната ни директория е последна версия – тоест дали някой друг не е направил промени по някой (някои) от файловете след последния път, когато сме теглили проекта или сме се update-вали. Ако такива промени има, те биват изтеглени от централния сървър и биват приложени върху нашата работна директория.
7. *Update to revision* – същото като горното, само че ние избираме до коя версия искаме да се ъпдейтнем. Полезно, ако примерно последната версия е счупена, и ние искаме да изтеглим някоя по-ранна, работеща. Всъщност има още много случаи, в които бихме искаме да дадем update to revision вместо само update, но със сигурност и сами ще се сблъскате с такива.
8. *Revert* – Премахва промените в работаната директория, които сме направили от последния update насам. Примерно сме се ъпдейтнали, написали сме някаква боза, която сме я омазали, искаме да я почнем наново – просто цъкаме revert и отново сме в изходна позиция.
9. *Get Lock* – „заключва" даден файл или множество от файлове. Когато даден файл е заключен, единственият човек, който може да го променя, е този, който го е заключил. Подходящо, ако много хора искат да променят едно и също нещо, но един е достатъчен. Тогава той може да направи lock на нещото, да го промени, и после да го отключи. Така ще е сигурен, че никой няма да го пипа докато той работи по него. Примерно ако файлът е картинка, и един от дизайнерите реши да я промени, е хубаво да я заключи,

понеже ако двама дизайнери променят 1 картинка после нейното „merge"-ване е почти невъзможно (във всеки случай много по-трудно).

10. *Release Lock* – логично, след като сме свършили с работата по даден заключен файл е хубаво да го отключим.
11. *Branch* – създава нов „клон" на проекта. Бранч-ването де факто е копиране на целия проект в друга директория, като веднага след branch-а двете копия са напълно идентични. Подходящо ако има два логични подхода за нататъшно разработване на проекта, и някои програмисти искат да направят единия, а други – другия. Доста често срещано при open source проектите. Също удачно ако някой иска да направи много голяма промяна – препоръчително е да работи върху собствен branch вместо върху общата версия. Например ако се разработва един продукт, който трябва да работи на няколко операционни системи, често има различни branch-ове за различните операционни системи.
12. *Merge* – сливане на две едновременни промени на двама програмисти върху един и същ сорс. Това е много много често срещано и може би най-трудното за правене. Представете си, че двама програмисти в един и същ ден (седмица, месец, whatever) са променяли един и същ файл по такъв начин, че някои от редовете са модифицирани както от единия, така и от другия. При commit и на двамата ще се получат така наречените конфликти (conflicts) които трябва да бъдат разрешени (resolved). Тоест някой (единият програмист, другият програмист, или трети човек) трябва да седне и за всеки от конфликтите да определи дали в repository-то ще се влезе версията на първия програмист, на втория програмист или трябва да се напише нов код за да може след сливането промените и на двамата да работят. Merge-ването често е доста отговорна и тежка работа, особено ако конфликтите са много (или пък са с тежка логика). Обикновено в началото като работите вие ще сте един от двамата програмиста, а някой друг ще трябва да merge-ва кода ви =)
13. *Create patch* – създава файл, който показва каква е разликата между вашата работна директория на проекта и версията, върху която сте работили (обикновено последната версия в repository-то, ако се update-вате достатъчно често). Това е подходящо в няколко случая. Първо, ако искате да пратите на някого вашата версия на сорса, без това да минава през SVN. Примерно, ако даден проект се администрира и вие нямате права за писане (тоест не можете просто да дадете commit), вие изпращате Patch на някой от админите (примерно някой review board, което ще разгледаме по-нататък в хода на курса). Администраторите на проекта разглеждат вашите промени и решават дали да ги добавят в общото repository или не. Patch-ът представлява текстов файл с направените от вас промени, който е лесно четим от SVN, тоест позволяващ автоматично прилагане на промените.
14. *Apply patch* – именно това автоматично прилагане на промените. Примерно ние сме администратор на даден проект, някой ни е изпратил patch, който ние решаваме да приложим към проекта. Давайки apply patch на неговия patch се прилагат промените му върху нашата работна директория. Оттам нататък ние можем да дадем commit, като така ще бъдат приложени в repository-то.

15. *Import* – Ако решим в даден проект да вмъкнем друг проект (обикновено по-малък такъв). Каква е разликата с това просто да добавим всички негови файлове? Губим историята (тоест неговите revision-и). Затова е доста по-добре да го Import-нем.
16. *Export* – Ако искаме да извадим всички смислени файлове на един проект, без meta информацията (hidden SVN файлове, които са нужни на SVN да поддържа разликите между нашата версия и тази в последния Update, където се пази URL-то на repository-то, и т.н.). Използва се, ако примерно искаме да дадем готовия код на човека (фирмата), който (която) ни е възложила проекта. На тях служебната информация не им трябва.
17. *Info* – Дава информация за номера на последно-изтеглената версия, върху която работим в момента, а също така и допълнителна информация за това кой е направил последната промяна (име на user), дата, на която е направена тази промяна и други. Ползва се сравнително често, особено ако имате няколко копия на репозиторито (с цел да работите по различни задачи върху един и същ код (или най-малкото проект)).
18. *Stat* – Дава информация за променените файлове (спрямо последната изтеглена версия – тоест версията, която би ви дала командата info) и по какъв начин са променени файловете. Например списъкът може да съдържа имената (с пътищата) до различни файлове, като до всеки от тях е зададено дали той е бил модифициран, изтрит, добавен, unversioned (тоест нов, но на които още не е дадено Add) или други.
19. Изключително вероятно е някои от тези да ги има в теста.

В часа също така (сравнително набързо) разгледахме как можем да измерваме време в C/C++. Тъй като това ще е нещо, което ще ползваме при разработката на проекта, ще можем да го демонстрираме в реален код и ще се върнем отново на него.
Бяха разгледани двете основни функции, които можем да ползваме. По-стандартната от тях е clock() (от библиотеката <time.h>), която ни връща броя „цъкания" на системния часовник от стартирането на нашата програма. Тези цъкания са приблизително всяка милисекунда, но техния точен брой в една секунда се задава чрез макроса CLOCKS_PER_SEC. За да измерим времето, изразходвано в даден фрагмент код, трябва да извикаме clock() преди и след фрагмента и да видим колко е разликата между тези два момента. Примерът по-долу показва как можем да измерим за колко време върви bubble sort на един масив от цели числа.

```
int n = 10; // Броят числа в масива
int a[10] = {42, 13, 7, 666, 1337, 5, 2, 2, -6, 10}; // Самият масив
unsigned startTime = clock(); // Измерваме времето преди изпълнението
for (int i = 0; i < n; i++)
      for (int c = 0; c + 1 < n; c++)
            if (a[c] > a[c + 1]) swap(a[c], a[c + 1]);
unsigned endTime = clock(); // Измерваме времето след изпълнението
cout << "Execution time was " << (double)(endTime - startTime) /
(double)CLOCKS_PER_SEC << " seconds" << endl;
```

За съжаление системният часовник не е особено прецизен. Примерно няколко последователни викания на clock() (без нищо между тях) биха върнали един и същ резултат. Нещо повече, скоковете между върнати резултати от clock() са приблизително през 15 милисекунди. Тоест ако просто в един фор цикъл викаме clock() и не правим нищо друго, първите n викания ще върнат резултат x, а n+1-вото викане ще върне x+15 (примерно). В някои нови процесори/операционни системи часовникът е с около 1000 пъти по-точен (тоест връща микросекунди, а не милисекунди).
Друга алтернатива за измерване на време (единствено под Windows, така че не е за предпочитане) е GetTickCount(), която пък връща броя милисекунди от стартирането на операционната система. Тя е дефинирана в хедъра <windows.h>.

Втори час
Теоретични задачи от интервюта

1. Даден е масив с N числа, като всяко от тях е между 1 и N. Измислите алгоритъм, който проверява дали има повтарящо се число, като ползва O(1) допълнителна памет и O(N) време.

2. Даден е масив с 2 * N + 1 числа. Всяко число се среща по два пъти с изключение на едно, което е само. Намерете това число.
Разгледайте алгоритми със сложност O(N**2), O(N * logN), и O(N), както и O(N) време и O(1) памет.

3. Ели е една страхотна домакиня. Без нея домашният уют не би бил дори близо до това, което е сега. Това се дължи главно на нейните манипулативни способности - тя някак си успя да накара Крис да сготви и изчисти, а Станчо да изпере и простре. Станчо, недоволен от възложената му задача, я вършеше с неохота, докато пиеше минерална вода. Вече беше стабилно подпийнал (второ шише) и докато простираше успя да изпусне два различни чорапа на Ели през балкона.
Помогнете на Ели да намери кои са липсващите чорапи! Всички чорапи в началото са били по двойки, и можем да представим като естествени числа, като различни чорапи са представени с различни числа.

4. От всеки от трите ъгъла на триъгълник с равен шанс се избира вектор към някой от другите два ъгъла. Каква е вероятността два от тези вектори да са срещуположни?