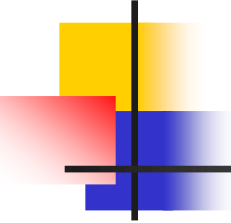


Configuration Management (CM)

- Przykłady praktyk, narzędzi, wskazówek, ...

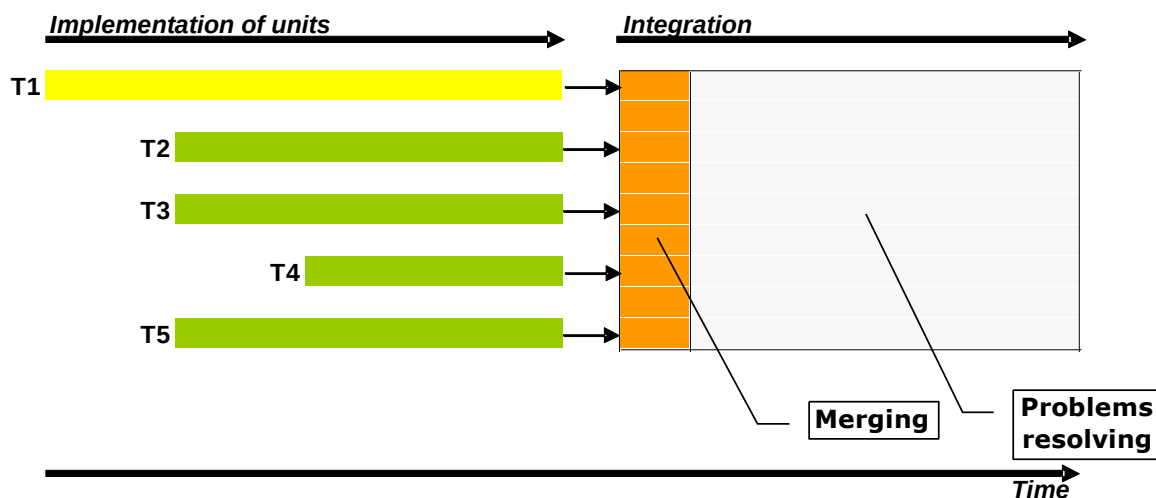


Zagadnienia

- Daily Build & Smoke Test
- System build – przykład narzędzia
- CM w praktyce – jak to działa

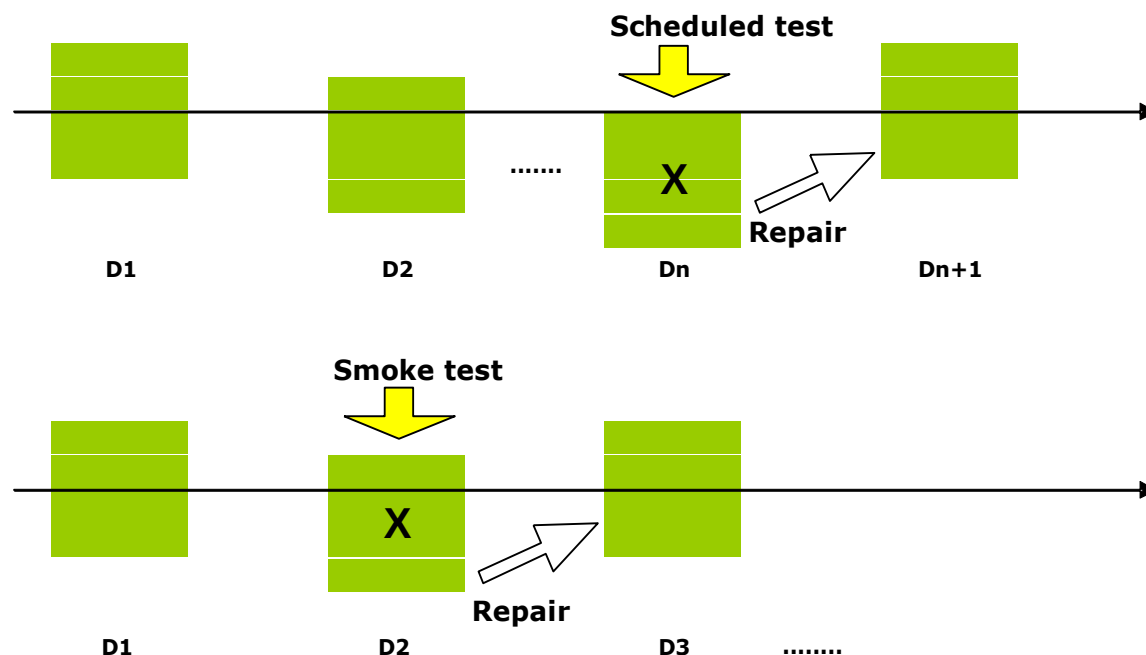
Daily Build and Smoke Test - zalety

- Minimalizacja ryzyka integracji
 - Podstawowe ryzyko integracji – działające osobno komponenty nie współpracują razem
 - Większość czasu integracji – debugging, re-design, re-implementacja
 - W skrajnych przypadkach – zamknięcie projektu



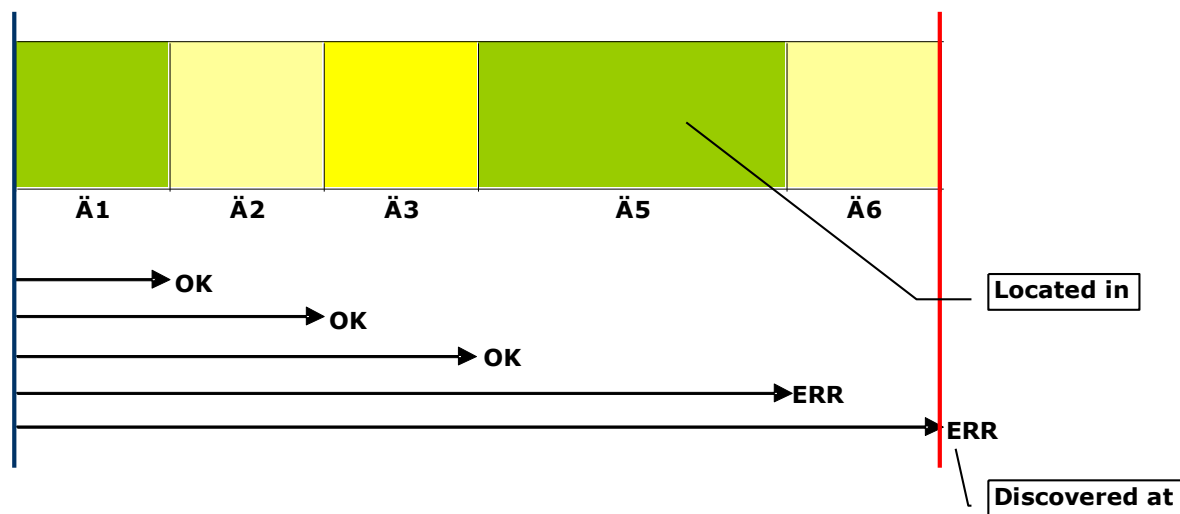
Daily Build and Smoke Test - zalety

- Zmniejszenie ryzyka niskiej jakości
 - Ryzyko niskiej jakości jest powiązane z ryzykiem wystąpienia problemów integracyjnych
 - Codzienne (automatyczne) testy – zmniejszenie ryzyka istotnego odchylenia od planowanej funkcjonalności
 - Po każdorazowym wykryciu defektu(ów) – przywrócenie systemu do stabilnego stanu



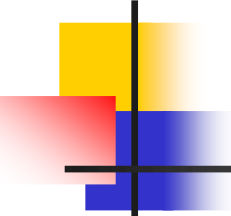
Daily Build and Smoke Test - zalety

- Ułatwienie diagnozowania oraz lokalizacji defektów
 - System buildowany i testowany codziennie
 - Łatwe wychwycenie momentu wprowadzenia błędu
 - Np. poprawny wynik testów w dniu 16 a w 17 już nie – błąd wprowadzony w obrębie kodu zmodyfikowanego w dniu 16



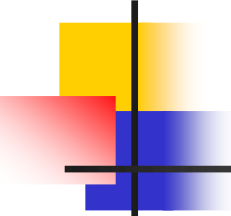
Daily Build and Smoke Test - zalety

- Podniesienie morale zespołu
 - Możliwość bieżącej obserwacji jak system się rozwija
 - Kapitalne znaczenie dla morale
 - Nie ma większego znaczenia CO system robi (jaka jest jego funkcjonalność)
 - Nawet wyrysowanie prostokąta na ekranie..  cieszy
- Ale... 
 - DB&ST nie jest odpowiednikiem procesu testowania systemu, jedynie jego uzupełnieniem!!
 - Smoke testy – w miarę proste ze względu na koszt ich wytworzenia (test data, test cases, itp.)



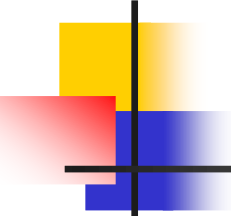
Daily Build and Smoke Test - zasady

- Codzienne buildowanie systemu
 - Proces buildowania – “puls” projektu
 - Pomiedzy kolejnymi uderzeniami pulsu
 - Kod tworzony przez różnych programistów
 - Ma prawo się „rozjechać”
 - W momencie kolejnego uderzenia
 - Kod należy ze sobą zsynchronizować
 - Przy położeniu nacisku na częste buildowanie
 - Zapobiega się całkowitej utracie synchronizacji pomiędzy wersjami kodu różnych członków zespołu



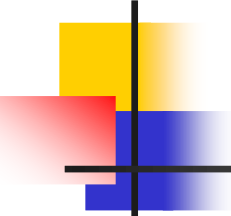
Daily Build and Smoke Test - zasady

- Sprawdzanie i analiza nieudanych buildów
 - Każdy kolejny build musi zostać zaliczony do „udanych”
 - Jeśli nie – poprawienie systemu → najwyższy priorytet !
- Co oznacza „udany build” ?
 - Kompilacja wszystkich plików, bibliotek oraz pozostałych komponentów – zakończona powodzeniem
 - Linkowanie wszystkich plików, bibliotek oraz pozostałych komponentów – zakończone powodzeniem
 - Brak błędów uniemożliwiających uruchomienie programu
 - Np. zła konfiguracja plików startowych
 - Przejście „smoke testów” →



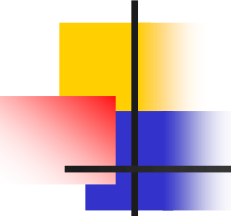
Daily Build and Smoke Test - zasady

- Codzienne przeprowadzanie „smoke testów”
 - Testy te powinny mieć całościowy charakter
 - Wszystkie kluczowe funkcjonalności
 - Niezbyt dogłębnie – przejście ST pozwala na dalsze, szczegółowe testowanie
 - Bez ST – codzienne buildowanie traci sens
 - Staje się tylko dowodem faktu poprawnej kompilacji
 - W miarę rozwoju systemu ST muszą również ewoluować
 - Początkowa postać – może być np. „Hello, World!”
 - Dochodzą kolejne zaimplementowane funkcjonalności
 - Czas trwania testów: początkowo – kilka sekund, potem zwiększa się to do 30 min,



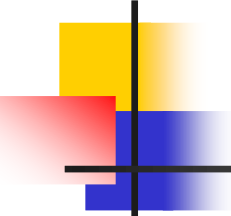
Daily Build and Smoke Test - zasady

- Utworzenie zespołu zajmującej się procesem DB&ST – tzw. *build group*
 - W większości projektów obsługa codziennego buildowania systemu oraz aktualizacja smoke testów → jawny przydział pracy (cały/pół etatu)
- Przykład: MS Windows NT 3.0
 - Build group składająca się z 4 osób przydzielonych na cały etat do tego zadania



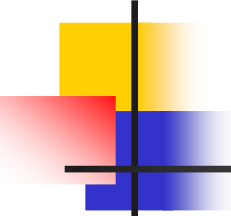
Daily Build and Smoke Test - zasady

- Dodawanie kolejnych części systemu do buildu
 - Tylko wtedy kiedy jest to sensowne !
 - Zaimplementowana funkcjonalność – spójna, przechodzi unit-testy
 - Szybkość pisania poszczególnych osób
 - Znaczące zmiany – zwykle co parę dni
 - Nanoszone – tylko w spójnym stanie



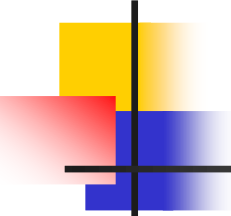
Daily Build and Smoke Test - zasady

- Kary za „rozwalenie” buildu
 - Podstawa: udany build – wysoki priorytet projektu
 - „Wywalony” build – wyjątek (nie zasada!)
 - „Wywalony” build – cała bieżąca praca autora zamrożona aż do usunięcia bug-u
 - Zbyt częste nieudane buildy – sensowność całej procedury - ??



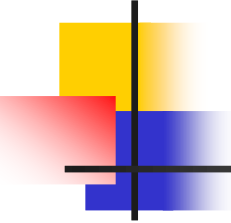
Daily Build and Smoke Test - zasady

- Kary za „rozwalenie” buildu
 - Często stosowane praktyki – próba podejścia ambicjonalnego
 - Nie jest to „kara” stricte
 - Winowajca - ...
- Przykład: Microsoft ...
 - Windows NT
 - Windows 95
 - Excel
- ➔ beepery dla developerów



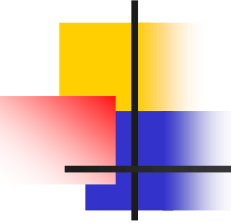
Daily Build and Smoke Test - zasady

- W warunkach nacisku czasowego – również stosuj DB&ST
- Nakład na DB&ST w takich warunkach – zbędna ekstrawagancja ?
- Na odwrót – w tego typu sytuacjach
 - „Skróty” w designie i kodzie
 - Mniej uwagi na unit-testy i review
 - Kod – bardziej chaotyczny
- DB&ST
 - Wczesne wykrywanie konsekwencji powyższych czynników



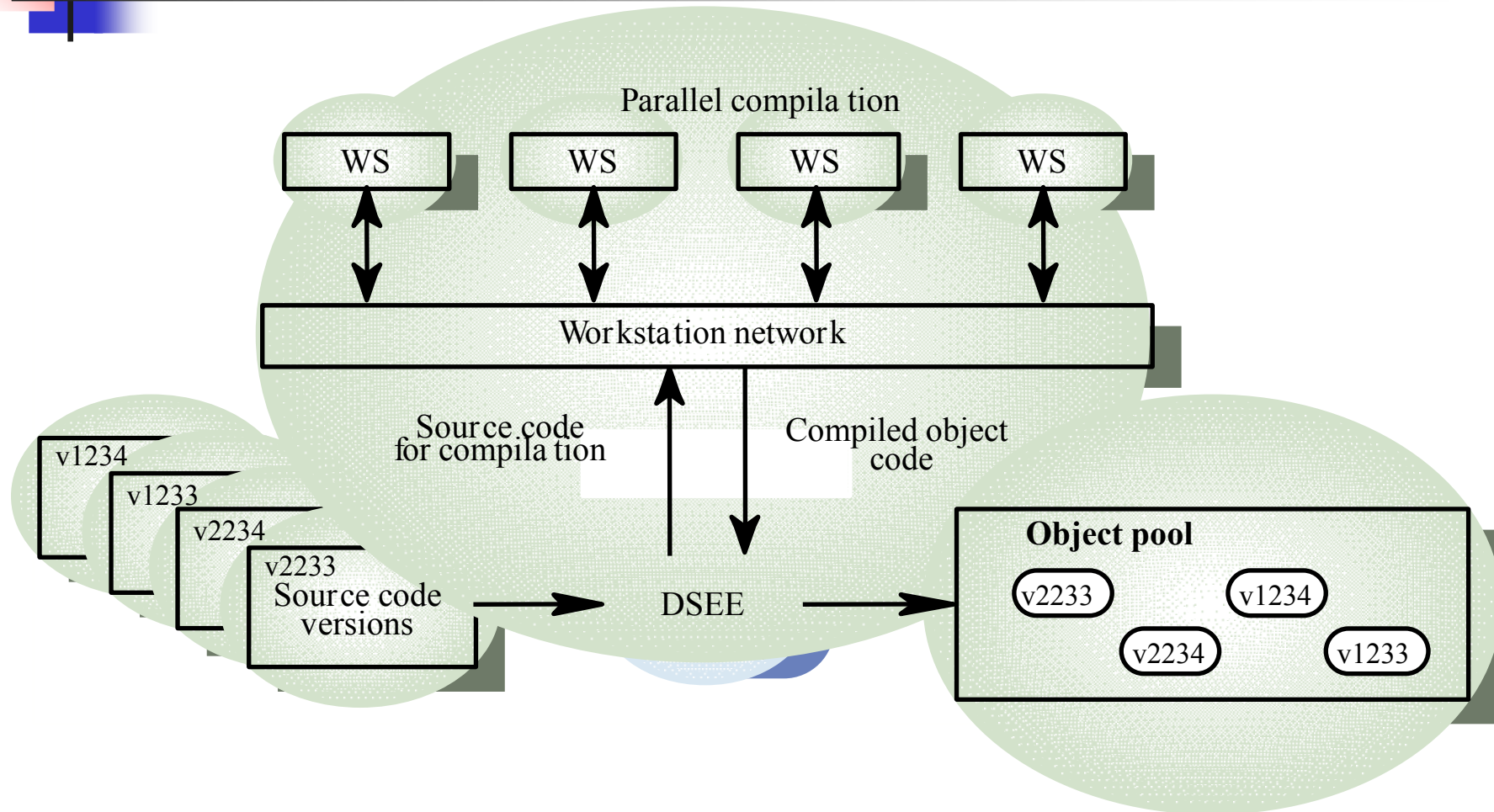
Daily Build and Smoke Test - podsumowanie

- Kiedy tego używać?
 - Projekt zbyt duży – codziennie buildowanie bez sensu ?
 - Ale ... znane duże projekty stosowały to z powodzeniem !
 - Np. MS Windows 3.0
 - 5.6 milionów linii kodu
 - 40 000 plików z kodem źródłowym
 - Cały build: 19 godzin na kilku maszynach
 - DB&ST był stosowany konsekwentnie – cały czas życia projektu
 - Mniejsze projekty ➔ łatwiejszy build
- Ogólnie: dobra praktyka!
 - Mając na uwadze: NIE jest to substytut testów!!



System build – przykład narzędzia

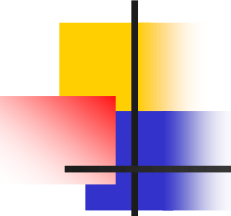
- Proces buildowania systemu – czasochłonny
- Wymagane zasoby pamięciowe i obliczeniowe
- Idea – wykorzystanie maszyn w sieci
 - Większość maszyn w sieci – małe obciążenie
- DSEE
 - Hewlett-Packard (Leblang and Chase, 1987; Lubkin, 1991)
- Cechy
 - Równoległe generowanie różnych wersji systemu
 - Równoległa kompilacja kodu na różnych maszynach w sieci
 - Kontrola wersji kodu



- Zintegrowane procesy kontroli wersji oraz buildowania systemu
- Identyfikacja składowych systemu
 - Tradycyjna: które pliki generują system
 - Tzw. configuration thread – określa które wersje plików, narzędzi oraz jakie opcje kompilacji generują daną wersję systemu
- Zarządzanie obiektami pochodnymi
 - Zachowywane w obrębie puli (ang. *object pool*) możliwie jak najdłużej
 - Obiekty pochodne powiązane z odp. wersjami plików źródłowych

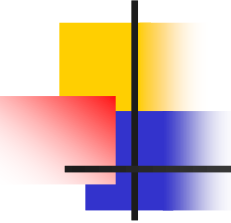
- Brak automatycznej rekompilacji po zmianie kodu źródłowego (jak to ma miejsce w Make)
- Rekompilacja – w momencie zażądania danej wersji pliku pośredniego
 - a ta jest nieaktualna (odp. wersja kodu po zmianie)
- Identyfikacja wersji – oparta o atrybuty
- Kod obiektowy powiązany z kodem źródłowym poprzez atrybuty. Skompilowany plik pośredni dziedziczy atrybuty od swojego pliku źródłowego
 - Przykłady atrybutów: numer wersji (15), identyfikator („R3”), stan („beta-release”), ...

- Wykorzystanie maszyn pracujących w sieci
 - maszyny wolne/mało obciążone przez większość czasu
- Lokalizacja nieobciążonej maszyny, polecenie kompilacji komponentu (ów)
 - Skompilowane elementy trafiają do *object pool*
- Wzrost wydajności – rzędu kilkuset %
- Im bardziej homogeniczne maszyny w sieci tym lepiej pracuje system
 - poszczególne komponenty stają się wymienne



CM – w praktyce

- Przykładowe narzędzie: Rational CC
- Pojęcia: VOB, view
- Cecha: wszyscy pracują na VOB-ie
 - Odpada konieczność kopiowania plików z/do katalogu roboczego
 - Elementy VOB-a są widoczne dla wszystkich użytkowników
 - Elementy *view* tworzone przez danego użytkownika – widoczne tylko dla niego



CM – w praktyce

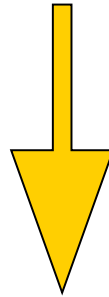
- Specyfikacja konfiguracji

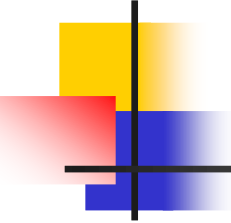
- Ogólna postać

`scope pattern version-selector [optional-
clause]`

- Przykład

`element * CHECKEDOUT
element * VER_1.0.2
element * /main/LATEST`





CM – w praktyce

```
ls foo.c
```

```
> ls foo.c  
-r-xr-xr-x 12 Nov 02 11:50 1743 foo.c
```

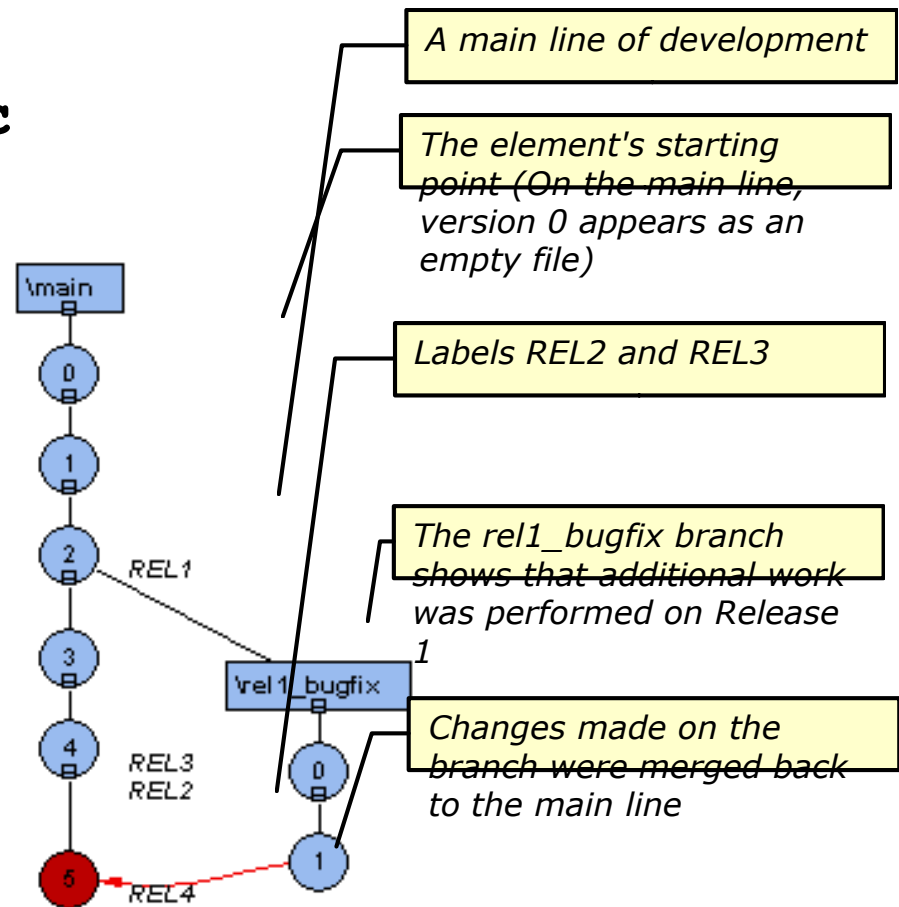
```
cleartool ls foo.c
```

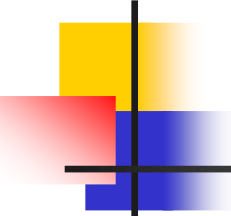
```
> cleartool ls foo.c  
foo.c      /main/5      (ELEMENT * /main/LATEST)
```

CM – w praktyce

cleartool lsvtree foo.c

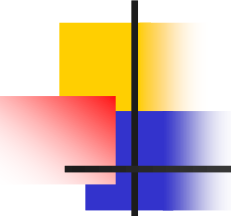
```
> cleartool lsvtree foo.c
foo.c@@/main
foo.c@@/main/0
foo.c@@/main/1
foo.c@@/main/2 (REL1)
foo.c@@/main/3
foo.c@@/main/4 (REL2, REL3)
foo.c@@/main/5 (REL4)
foo.c@@/main/rel1_bugfix
foo.c@@/main/rel1_bugfix/1
```





CM – w praktyce

- Podczas pracy wielu osób nad kodem
 - Rezerwacja pliku – tzw. checkout
 - Możliwa równoczesna przez parę osób
 - Po wprowadzeniu zmian – konieczne jest ich scalenie (*ang. merging*)
 - Analogicznie – przy pracy na oddzielnych gałęziach (*ang. branches*)



CM – w praktyce

- Praca na gałęziach
 - Różne moduły projektu
 - Bugfix dla błędu wprowadzonego do wcześniejszego release-u
 - Większa (wymagająca kilku operacji check-in) zmiana kodu przez 1 osobę
- Scalenie ➔

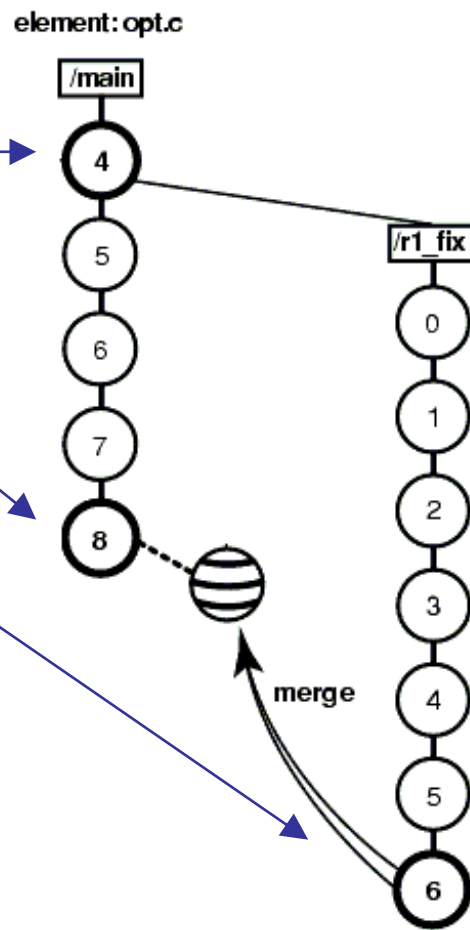
CM – w praktyce

■ Scalenie →

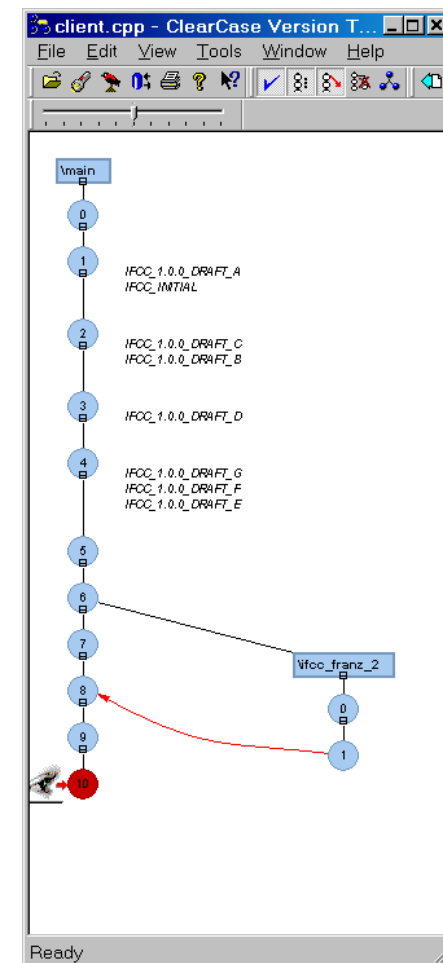
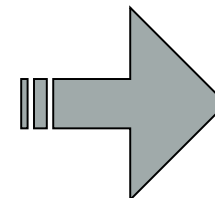
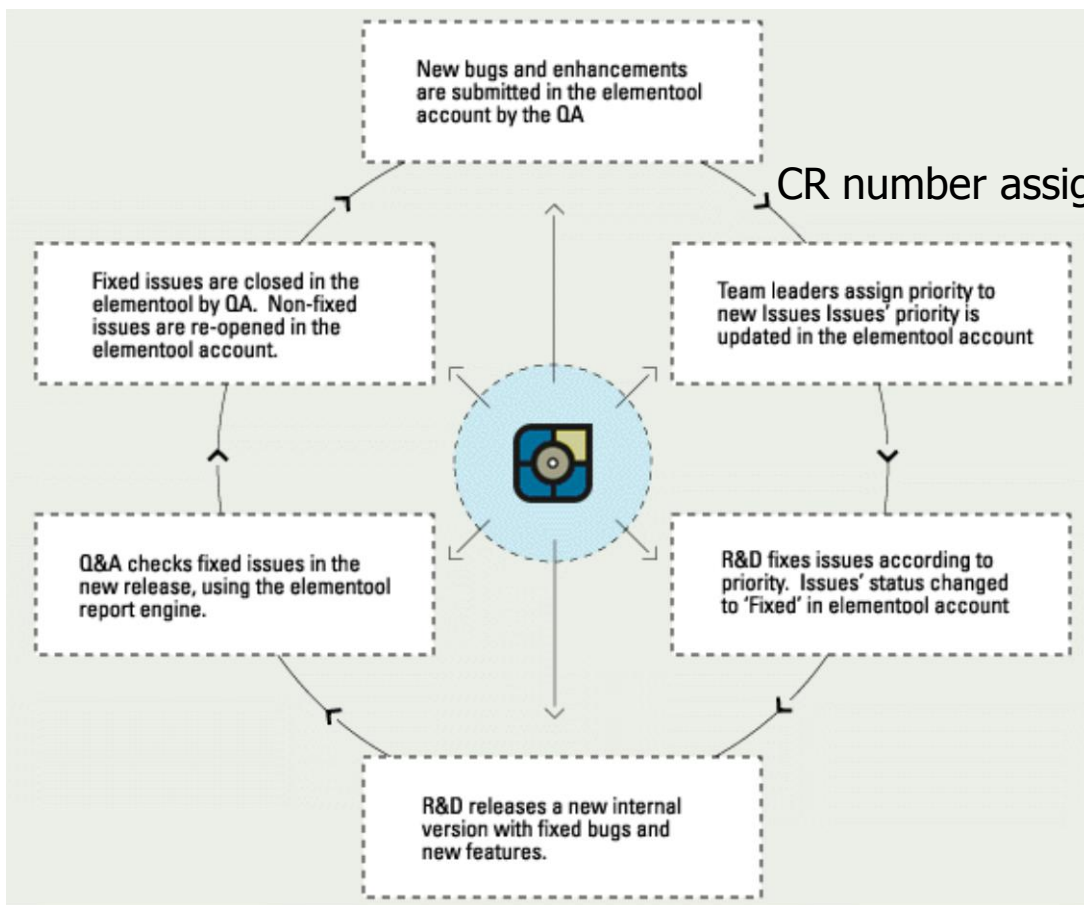
- Base Contributor
- Target Contributor
- Contributor(s)

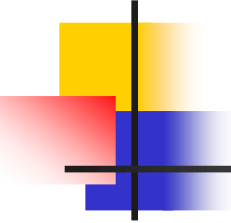
■ Algorytm

- Linie pliku
 - niezmienione
 - zmienione w 1 C
 - zmienione w > 1 C



CM – w praktyce





Podsumowanie

