အခန်း(၅) – Version Control System

"ကဲ... Demo လုပ်ဖို့ အားလုံးအသင့်ပဲလားလေ့။"

"ဟုတ်ကဲ့.. နေ့လည်လောက်ဆိုရင်တော့ အသင့်ဖြစ်ပါပြီ။"

"ဘယ်လိုဖြစ်တာလဲ။ မနေ့ကတည်းက အားလုံးပြီးနေပြီးသား မဟုတ်လား။"

"ဟုတ်ပါတယ်။ Code တစ်ချို့ပြင်လိုက်မိတာ အလုပ်မလုပ်တော့လို့ ပြန်စစ်နေတာခင်ဗျ။"

"ဟာ... ပြဿနာပဲကွာ။ ဘာလို့ခုချိန်မှပြင်တာလဲ။ ဒီနေ့ Demo ပြမယ်လို့ကြေငြာပြီးနေပြီ။ နဂို အလုပ်လုပ်နေတဲ့အတိုင်း ပြန်ထားလိုက်လေ။"

"ဟုတ်ကဲ့... နဂိုအတိုင်းပြန်ထားပြီးပါပြီ။ ဒါလည်း အလုပ်မလုပ်သေးဘူး။ ဘာဖြစ်သွားလဲ မသိဘူး။"

"ခက်တာပဲကွာ။ တော်သေးတာပေါ့၊ အရင် Version တွေ ငါသိမ်းထားမိလို့။ Share Folder ထဲ ကိုဝင်ပြီး အရင် Version ကို ကူးယူလိုက်။"

"Share Folder ထဲမှာတွေ့ပါတယ်။ ဒါပေမယ့် project-latest, project-update, project-final, project-stable, project-update-final, project-real-latest အများကြီးဖြစ်နေတယ်။ အဲ့ဒီထဲက ဘယ်ဟာကို ကူးယူရမလဲ"

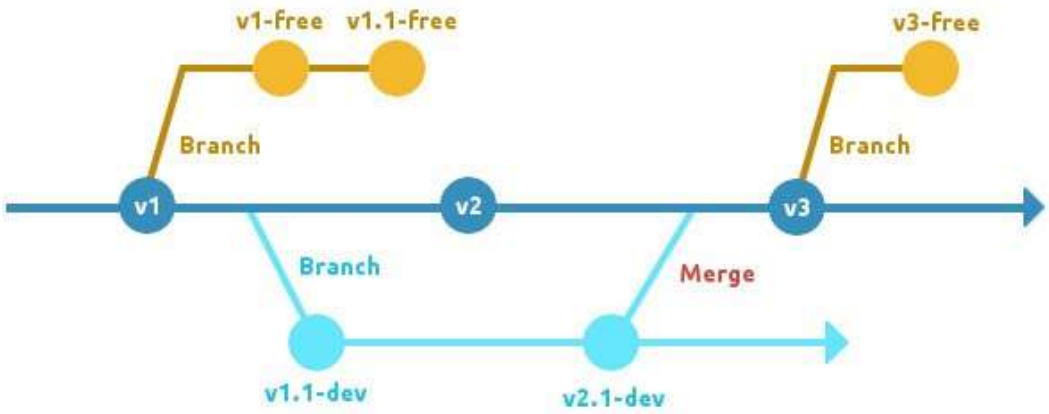
"project-latest ကိုယူလေကွာ။ အဲ... နေဦး။ project-real-latest ဖြစ်မယ်ထင်တယ်။ ရှုပ်တယ်ကွာ ဘယ်ဟာနောက်ဆုံး Version လဲသေချာအောင်၊ တစ်ခုချင်းသာ လိုက်စမ်းကြည့်လိုက်တော့။"

ဒါဟာ Version Control System အသုံးမပြုတဲ့ Software Team တွေမှာ ကြုံတွေ့ရလေ့ရှိတဲ့ ပြဿနာတစ်ရပ်ပဲ ဖြစ်ပါတယ်။ Software Project တစ်ခုဆိုတာဟာ အချိန်နဲ့အမျှ ပြောင်းလဲနေတက်တဲ့သဘာဝ ရှိပါတယ်။ ရှေ့ပိုင်းမှာလည်း၊ တိကျတဲ့ Requirement ရရှိဖို့ ခက်ခဲတက်ကြောင်းနဲ့၊ အလုပ်လုပ်နေတဲ့ Code ကို တစ်နေရာမှာ ပြုပြင်ခြင်းအားဖြင့် အခြား တစ်နေရာမှာ မမျှော်လင့်ပဲ သက်ရောက်သွားပြီး Bug တွေ ဖြစ်ပေါ်တက်ကြောင်းတို့ကို ဆွေးနွေးခဲ့ပြီးဖြစ်ပါတယ်။ ဒီလို အခက်အခဲတွေ ရှိတက်လို့ဆိုပြီး ပြုပြင်မှုတွေကို မလုပ်ချင်လို့ မရပါဘူး။

အပြောင်းအလဲတွေရှိလာလို့ ပြုပြင်ဖြည့်စွက်ရတဲ့အခါမှာ၊ မမျှော်မှန်းထားတဲ့ အမှားတစ်ခုကြောင့် မူလအလုပ်လုပ်နေတဲ့ စနစ်ကို ထိခိုက်ပြီး အလုပ်မလုပ်တော့တာမျိုး ဖြစ်တက်ပါတယ်။ ဒါကြောင့် လက်ရှိအလုပ်လုပ်နေတဲ့ အဆင့်တစ်ခုကို Version တစ်ခုအဖြစ် သတ်မှတ်ပြီး၊ သီးခြားခွဲခြားသိမ်းဆည်းထားပြီးမှ ဖြည့်စွက်ပြင်ဆင်မှုတွေကို ဆက်လက်ဆောင်ရွက်ရတဲ့ သဘောရှိပါတယ်။ အဲ့ဒီလိုခွဲခြားထားမှလည်း၊ ဖြည့်စွက် Code တွေကို ရဲရဲဝံ့ဝံ့ ဖြည့်စွက်စမ်းသပ်နိုင်မှာဖြစ်ပါတယ်။ အသစ်ဖြည့်စွက်မှုကြောင့် အမှားအယွင်း တစ်စုံတစ်ရာ ရှိခဲ့ရင်လည်း မူလအလုပ်လုပ်နေတဲ့အခြေအနေကို အချိန်မရွေး ပြန်လည်ရရှိနိုင်မှာဖြစ်ပါတယ်။ Version ခွဲခြားမထားရင် မှားယွင်းမှုဖြစ်သွားမှာကို စိုးရိမ်စိတ်နဲ့ ပြင်ဆင်ဖြည့်စွက်မှုတွေကို ရဲရဲဝံ့ဝံ့ မလုပ်ရဲတော့တဲ့အတွက် အလုပ်မတွင်ဖြစ်တက်ပါတယ်။ ဒီထက်ပိုဆိုးတာက၊ ပြင်ဆင်ဖြည့်စွက်မှုကြောင့် အမှားအယွင်းရှိလာတဲ့အခါ မူလအခြေအနေ ပြန်ရောက်ဖို့ကို အတော်လေး အချိန်ပေးပြီး ပြန်လည်ရှင်းလင်းရတက်လို့ အချိန်တွေ ကုန်တက်ပါသေးတယ်။

ဒီလို Version တွေခွဲခြား မှတ်တမ်းတင်တဲ့ လုပ်ငန်းဆောင်ရွက်ရာမှာ၊ Source Code ဖိုင်တွေကို ကိုယ်တိုင် Copy တွေ ကူး၊ နာမည်အမျိုးမျိုးပေး စသဖြင့် Manual လုပ်နေစရာမလိုပဲ၊ စနစ်ကျထိရောက်အောင် ကူညီပေးနိုင်တာကတော့ Version Control System (VCS) တွေပဲဖြစ်ပါတယ်။ Version Control System ကို Revision Control System လို့လည်းခေါ်သလို၊ Source Code Management System (SCM) လို့လည်း ခေါ်တက်ကြပါသေးတယ်။

Version Control System တွေက Version တွေသာမက Branch လုပ်ဆောင်ချက်တွေလည်း ပါဝင်တက်ပါတယ်။ ဥပမာ - Code Base တစ်ခုတည်းကို အခြေခံပြီး၊ Free Version, Pro Version စသဖြင့် Branch များ ခွဲခြားဆောင် ရွက်နိုင်ခြင်းပဲ ဖြစ်ပါတယ်။



ပုံ (၅.၁) - Branching in VCS

ပုံ (၅.၁) မှာလေ့လာကြည့်ပါ။ v1, v2, v3 စတဲ့ Version တွေဟာ အဆင့်လိုက်မှတ်တမ်းတင်ထားတဲ့ မူလ Version တွေဖြစ်ပါတယ်။ ဒီမူလ Code Base ကနေ လုပ်ဆောင်ချက်အနည်းငယ်ကွဲပြားတဲ့ သီးခြား Version တွေဖန်တီးဖို့ အတွက် နောက်ထပ် Project တစ်ခု ဖွင့်ဖို့မလိုပါဘူး။ လက်ရှိ Code Base ကနေပဲ Branch အဖြစ် ခွဲထုတ်လိုက်ပြီး အတူတစ်ကွ ဆက်လက်ဆောင်ရွက်နိုင်စေမှာဖြစ်ပါတယ်။ မူလ Code Base ကို တစ်ချို့ VCS တွေက Trunk လို့ခေါ်ပြီး၊ တစ်ချို့ကတော့ master လို့ ခေါ်တက်ကြပါတယ်။

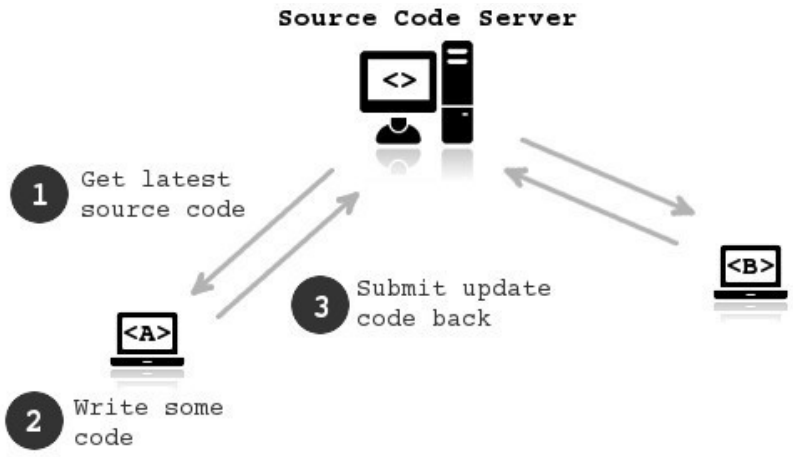
VCS တွေက master နဲ့ Branch တွေအကြား အပြန်အလှန် ကူးပြောင်းအလုပ်လုပ်နိုင်အောင် စီမံပေးထားကြပါတယ်။ master ကို ရွေးထားပြီး ပြင်ဆင်မှုတွေပြုလုပ်ရင် ပြုလုပ်လိုက်တဲ့ပြင်ဆင်မှုတွေက master Code Base ကိုပဲ သက်ရောက်စေမှာဖြစ်သလို၊ Branch တစ်ခုကိုရွေးထားပြီး ပြင်ဆင်မှုတွေပြုလုပ်ရင်တော့ ရွေးချယ်ထားတဲ့ Branch ကိုပဲ ပြင်ဆင်မှုက သက်ရောက်စေမှာ ဖြစ်ပါတယ်။

Branch တွေကို စမ်းသပ်လုပ်ဆောင်ချက်တွေ ထည့်သွင်းစမ်းသပ်ဖို့လည်း အသုံးပြုကြပါတယ်။ ဒီနည်းနဲ့ စမ်းသပ်လိုတဲ့ Code တွေက master Code Base ကို မထိခိုက်စေပဲ၊ သီးခြားခွဲထားတဲ့ Development Branch ပေါ်မှာ လွပ်လပ်လပ်လပ် စမ်းသပ်ရေးသားနိုင်စေမှာဖြစ်ပါတယ်။ Development Branch ပေါ်မှာ ရေးသားထားတဲ့ စမ်းသပ် လုပ်ဆောင်ချက်ကို မူလ Version မှာ ပါဝင်သင့်တယ်လို့ အတည်ပြုနိုင်တဲ့အခါမှာ အဲ့ဒီ Development Branch ကို master Code Base နဲ့ ပေါင်းစပ်လိုက်နိုင်ပါတယ်။ ဒီလုပ်ဆောင်ချက်ကို Merge လုပ်တယ်လို့ ခေါ်ပါတယ်။

ပုံ (၅.၁) မှာ ဖော်ပြထားချက်အရ၊ အလယ်က v1, v2, v3 စတဲ့ Version တွေဖြစ်ပြီး၊ အပေါ်က v1-free, v1.1-free စတဲ့ Version တွေကတော့ မူလ master Code Base ကနေ လုပ်ဆောင်ချက်တစ်ချို့ ပြောင်းလဲထားတဲ့ သီးခြား Version များအဖြစ် Branch ခွဲယူထားခြင်းဖြစ်ပါတယ်။ အောက်ဘက်က v1.1-dev, v2.1-dev စတဲ့ Version တွေက တော့ စမ်းသပ်လုပ်ဆောင်ချက်တွေ ထည့်သွင်းစမ်းသပ်နိုင်ဖို့ ခွဲခြားထားတဲ့ Branch ဖြစ်ပြီး အဲ့ဒီ Branch ကို မူလ master Code Base နဲ့ တစ်နေရာမှာ ပြန်လည် Merge လုပ်ထားတာကိုလည်း တွေ့ရမှာဖြစ်ပါတယ်။

VCS တွေက master နဲ့ Branch တွေပေါ်က မှတ်သားထားတဲ့ Version အမှတ်တိုင်းကို ပြန်သွားနိုင်အောင်လည်း စီမံ ထားပေးလို့ အမှားအယွင်းတစ်စုံတစ်ရာကြောင့်ပဲဖြစ်ဖြစ် အခြားအကြောင်း တစ်ခုခုကြောင့်ပဲဖြစ်ဖြစ် လိုအပ်လာတဲ့အခါ အရင် Version တွေကို အချိန်မရွေး ပြန်လည်ရယူနိုင်မှာပဲဖြစ်ပါတယ်။

ဒါတင်မက၊ အဖွဲ့လိုက်ပူးပေါင်းဆောင်ရွက်ရတဲ့အခါမှာလည်း Code Base တစ်ခုတည်းကနေ VCS အကူအညီနဲ့ ရယူရေး သားခြင်းအားဖြင့် Developer တစ်ဦးရေးသားထားတဲ့ Code နဲ့ နောက် Developer တစ်ဦးရေးသားထားတဲ့ Code ကို ပေါင်းစပ်တဲ့လုပ်ငန်းကို Manual လုပ်နေစရာမလိုတော့ပဲ VCS က စနစ်တစ်ကျပေါင်းစပ်ပေးသွားမှာပဲဖြစ်ပါတယ်။ ဒါကြောင့် VCS တွေကို Version တွေစီမံဖို့အတွက်သာမက အဖွဲ့လိုက်ပူးပေါင်းဆောင်ရွက်ရာမှာ အခြေခံ Tool တစ်ခု အဖြစ် အသုံးပြုတက်ကြပါသေးတယ်။ ဒါကြောင့်လည်း VCS တွေကို Source Code Management System (SCM) လို့ခေါ်ကြခြင်းဖြစ်ပါတယ်။ Version ကိုသာမက၊ Source Code နဲ့ ပက်သက်တဲ့ အခြားစီမံမှုလုပ်ငန်းများကိုပါ ဆောင်ရွက်ပေးတဲ့အတွက် ဖြစ်ပါတယ်။



ပုံ (၅.၂) - Centralized VCS Workflow

ဒီလိုအဖွဲ့လိုက်ပူးပေါင်းဆောင်ရွက်နိုင်ဖို့ Centralize Source Code Server လိုအပ်တက်ပါတယ်။ ဒီတော့မှ **ပုံ (၅.၂)** မှာ ဖော်ပြထားသလို၊ Team မှာပါဝင်တဲ့ Developer တွေက၊ အခြား Developer များရေးသားထားတဲ့ နောက်ဆုံးရလဒ်ကို Central Server ထံမှရယူခြင်း၊ မိမိတို့ရေးသားဖြည့်စွက်ချက်တွေကို Central Server ထံ ပြန်လည်ပေးပို့ခြင်းအားဖြင့် Code Base တစ်ခုတည်းကို Developer အများ ပူးပေါင်းဆောင်ရွက်ခြင်း လုပ်ဆောင်ချက်ကို ရရှိမှာပဲဖြစ်ပါတယ်။

5.1 VCS Terminologies

Version Control System တွေအများကြီးရှိပါတယ်။ စနစ်တစ်ခုနဲ့တစ်ခု အသေးစိတ်လုပ်ဆောင်ချက်တွေ မတူကြပေမယ့် အခြေခံရည်ရွယ်ချက်တူပြီး၊ ပါဝင်တဲ့ Function တွေနဲ့ လုပ်ငန်းစဉ် Operation တွေကို ခေါ်တဲ့အခေါ် အဝေါ်တွေလည်း တူတက်ပါတယ်။ VCS တစ်ခုမှာပါဝင်တက်တဲ့ အသုံးအနှုံးတွေကို ဖော်ပြပေးလိုက်ပါတယ်။

revision/version – Source Code ကို ပြင်ဆင်မှုတစ်ချို့လုပ်ပြီး မှတ်တမ်းတင်သိမ်းဆည်းလိုက်တဲ့ အမှတ်တစ်ခုကို version (သို့မဟုတ်) revision လို့ ခေါ်ပါတယ်။

branch – အတိုင်းအတာတစ်ခုထိ ပြီးမြောက်ပြီးသား Software Project တစ်ခုရဲ့ Source Code ကို ပြင်ဆင်ဖြည့်စွက်မှုတွေ ပြုလုပ်လိုတဲ့အခါ၊ မူလ Code Base ပေါ်မှာ တိုက်ရိုက်လုပ်မယ့်အစား သီးခြား branch တစ်ခုအဖြစ် မူပွားယူပြီးမှ ဆောက်ရွက်ကြလေ့ရှိပါတယ်။ ဒီနည်းနဲ့ စမ်းသပ်မှုကြောင့် ဖြစ်ပေါ်လာတဲ့ မလိုလားအပ်တဲ့ ပြဿနာတွေ မူလ master Code Base ကို သက်ရောက်ခြင်း မရှိတော့မှာပါ။ သီးခြား branch ပေါ်မှာ ဆောင်ရွက်နေတဲ့ ဖြည့်စွက်ပြင်ဆင်ချက်ဟာ လက်ခံနိုင်တဲ့ အတိုင်းအတာတစ်ခု ရောက်ပြီဆိုတော့မှ အဲ့ဒီ branch ကို မူလ master Code Base နဲ့ ပေါင်းစပ်စေခြင်း အားဖြင့် master Code Base ထဲမှာလည်း ဖြည့်စွက်ပြင်ဆင်ချက်အသစ်တွေ ရောက်ရှိပါဝင်သွားစေမှာ ဖြစ်ပါတယ်။

fork – fork ဆိုတာ branch လိုပဲ မူလ Code Base ကို မူပွားယူခြင်းဖြစ်ပါတယ်။ ဒါပေမယ့်၊ မူလ Code Base နဲ့ ပြန်လည်ပေါင်းစပ်ဖို့မရည်ရွယ်ပဲ သီးခြား Project တစ်ခုအနေနဲ့ ဆောင်ရွက်ဖို့ ရည်ရွယ်ချက်နဲ့ မူပွားယူတဲ့ သဘောမျိုးဖြစ်ပါတယ်။

master/trunk – နောက်မှ ခွဲယူထားတဲ့ branch မူကွဲမဟုတ်ပဲ၊ မူလစတင်ကတည်း အခြေပြုထားတဲ့ မူလ Code Base ကို master branch (သို့မဟုတ်) trunk လို့ခေါ်ပါတယ်။

merge – branch တစ်ခုနဲ့ ခွဲခြားစီမံနေတဲ့ Code Base ကို master (သို့မဟုတ်) အခြား branch တစ်ခုခုနဲ့ ပေါင်းစပ်စေတဲ့ လုပ်ဆောင်ချက်ဖြစ်ပါတယ်။ branch တစ်ခုတည်းမှာပဲ Developer တစ်ဦးရဲ့ပြင်ဆင်ချက်ကို အခြားတစ်ဦးရဲ့ ပြင်ဆင်ချက်နဲ့ ပေါင်းစပ်ခြင်းလုပ်ငန်းကိုလဲ merge လုပ်တယ်လို့ပဲ ခေါ်ပါတယ်။

commit/checkin – ပြင်ဆင်ဖြည့်စွက်မှုတွေပါဝင်တဲ့ Code ကို Version သစ်တစ်ခုအဖြစ် မှတ်တမ်းတင်လိုက်တဲ့ လုပ်ဆောင်ချက်ကို commit လုပ်တယ် (သို့မဟုတ်) checkin လုပ်တယ်လို့ခေါ်ပါတယ်။

change/diff – ပြင်ဆင်ထားတဲ့ Code နဲ့ နောက်ဆုံး commit လုပ်ခဲ့တဲ့ Version တို့အကြား ကွာခြားသွားတဲ့ ကွာခြားချက်ကို change (သို့မဟုတ်) diff လို့ခေါ်ပါတယ်။ ကွားခြားသွားတယ်ဆိုတာ အသစ်တိုးလာတာ လည်း ဖြစ်နိုင်ပါတယ်၊ မူလ Code ကို ပြင်လိုက်တာလည်း ဖြစ်နိုင်ပါတယ်၊ မူလ Code အချို့ကို ဖယ်ထုတ် လိုက်တာ လည်း ဖြစ်နိုင်ပါတယ်။

delta compression – Version အသစ်တစ်ခုအဖြစ် မှတ်တမ်းတင်သိမ်းဆည်းတဲ့အခါ Code Base တစ်ခုလုံးကို နောက်တစ်ကြိမ် ထပ်မသိမ်းတော့ပဲ၊ ကွားခြားသွားတဲ့ diff ကိုသာ သိမ်းဆည်းတဲ့လုပ်ဆောင်ချက်ကို delta compression လို့ခေါ်ပါတယ်။

change list/patch – ဥပမာ – Open Source Project တစ်ခုကို fork လုပ်ယူပြီး ကိုယ်ပိုင်ဖြည့်စွက်မှုတွေ လုပ်တယ် ဆိုပါစို့။ ရရှိလာတဲ့ ပြင်ဆင်ဖြည့်စွက်မှုမှတ်တမ်း Version ကို မူလ Project မှာပါဝင်သွားစေလိုတဲ့အခါ – မိမိ ပြုလုပ်ထားတဲ့ ပြင်ဆင်ဖြည့်စွက်မှုမှတ်တမ်းကို change list (သို့မဟုတ်) patch အဖြစ်ထုတ်ယူရပါတယ်။ ရရှိလာတဲ့ patch ဖိုင်ကိုမူလ Project ရဲ့ တာဝန်ခံထံ ပေးပို့နိုင်ပါတယ်။ တာဝန်ခံက စီစစ်လေ့လာပြီး ပေါင်းစပ်ပေးမယ်ဆိုရင် မိမိပြုလုပ်ထားတဲ့ ပြင်ဆင်ဖြည့်စွက်မှုတွေလည်း မူလ Project မှာ ပါဝင်သွားမှာဖြစ်ပါတယ်။

repository – Version အားလုံးနဲ့ Branch အားလုံးတို့ကို စုစည်းသိမ်းဆည်းထားတဲ့ Code Base ကြီးကို repository လို့ ခေါ်ပါတယ်။

checkout – Central Source Server ကနေ၊ နောက်ဆုံး Update Version (သို့မဟုတ်) Version တွေ ထဲကတစ်ခုကို ကိုယ့် Local Machine ထဲ ကူးယူတဲ့ လုပ်ဆောင်ချက်ကို checkout လုပ်တယ်လို့ခေါ်ပါတယ်။ တစ်ချို့ VCS တွေမှာ တော့ branch တစ်ခုကနေ နောက်တစ်ခုကိုကူးပြောင်းတဲ့ လုပ်ဆောင်ချက်အတွက်လည်း checkout ဆိုတဲ့ အသုံးအနှုံးကို ပဲသုံးပါတယ်။

working copy – Central Source Server ကနေ၊ checkout လုပ်ယူလိုက်လို့ ကိုယ့် Local Machine ထဲ ရောက်ရှိလာတဲ့ Version ကို working copy လို့ခေါ်ပါတယ်။

clone – clone ဆိုတာကတော့ Central Server ပေါ်က repository ကြီးတစ်ခုလုံးကို ကိုယ့် Local Machine မှာ မူပွား ယူလိုက်တဲ့ လုပ်ဆောင်ချက်ဖြစ်ပါတယ်။ checkout နဲ့မတူတာက၊ checkout က Version တစ်ခုကိုသာ ရယူခြင်းဖြစ်ပြီး၊ clone က Version တွေ branch တွေအားလုံးပါဝင်တဲ့ repository တစ်ခုလုံးကို ရယူခြင်းဖြစ်ပါတယ်။

conflict – Local Copy ပေါ်မှာ ပြင်ဆင်ဖြည့်စွက်မှုတစ်ချို့ လုပ်လိုက်တယ်ဆိုပါစို့။ အဲ့ဒီ ပြင်ဆင်မှုကို Central Server ထံမပေးပို့မီ Server က နောက်ဆုံး Version ကို အရင်ရယူရတဲ့သဘော ရှိပါတယ်။ ဒီလိုရယူလိုက်တဲ့အခါ ကိုယ်ပြင်ဆင် ထားတဲ့ Code ဖိုင် နဲ့ Server ကနေရရှိလာတဲ့ နောက်ဆုံး Version ရဲ့ Code ဖိုင် တူညီနေတဲ့အခါ conflict ဖြစ်ပါတယ်။ VCS က ကိုယ်ပြင်ထားတာကို နောက်ဆုံးအနေနဲ့ မှတ်ယူပေးရမလား၊ Server ကနေရရှိလာတဲ့ ပြင်ဆင်ချက်ကို နောက် ဆုံးအနေနဲ့ မှတ်ယူပေးရမလား မဆုံးဖြတ်နိုင်တဲ့အတွက် conflict ဖြစ်ရခြင်းဖြစ်ပါတယ်။ ဒီလို conflict တွေရှိလာတဲ့ အခါ ကိုယ်တိုင်စီစစ်ပြီး ပေါင်းစပ်၍သော်လည်းကောင်း၊ နှစ်ခုထဲက တစ်ခုကို နောက်ဆုံးအဖြစ် ဆုံးဖြတ်ပေး၍သော် လည်းကောင်း **resolve** လုပ်ပေးရတက်ပါတယ်။

head – HEAD လို့ သုံးကြလေ့ရှိပြီး နောက်ဆုံး commit, နောက်ဆုံး Version ကို ညွှန်းဆိုတဲ့ Keyword ဖြစ်ပါတယ်။

tag/label – commit လုပ်ပြီး မှတ်တမ်းတင်ခဲ့တဲ့ Version ပေါင်းများစွာရှိနိုင်ပါတယ်။ အဲ့ဒီထဲကမှ အရေးပါတဲ့ Version တွေကို အမည်ပေးတဲ့လုပ်ငန်းကို tag (သို့မဟုတ်) label လို့ခေါ်ပါတယ်။ ဥပမာ – အဆင့်ဆင့် သိမ်းဆည်းလာတဲ့ Version တွေထဲက Version အမှတ်တစ်ခုကို 0.1.0 လို့ tag လုပ်ထားနိုင်ပါတယ်။

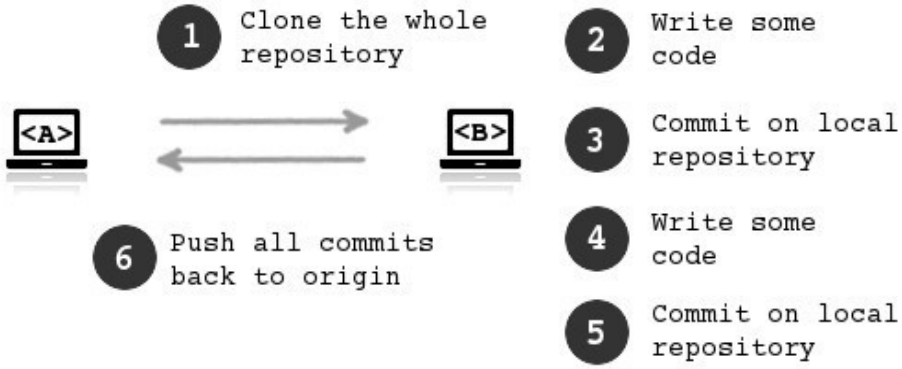
Version နံပါတ် ပေးပုံပေးနည်းကို မကြာခင်မှာ ဆက်လက်ဖော်ပြသွားပါမယ်။

5.2 – Centralized VCS vs. Distributed VCS

Version Control System တွေမှာ အမျိုးအစားအားဖြင့် နှစ်မျိုးရှိပါတယ်။ တစ်မျိုးကို Centralized Version Control System လို့ခေါ်ပြီး နောက်တစ်မျိုးကိုတော့ Decentralized Version Control System (သို့မဟုတ်) Distributed Version Control System လို့ ခေါ်ပါတယ်။

Centralized VCS တွေမှာ Source Code Repository ကို Central Server တစ်လုံးမှာထားပြီး Developer တွေက ကိုယ့် Local Machine ပေါ်မှာ Working Copy ကို ရယူအလုပ်လုပ်ကြရပါတယ်။ ဆိုလိုတာက၊ Branch တွေနဲ့ Version မှတ်တမ်းဟာ Central Server မှာသာရှိနေပြီး Developer တွေက နောက်ဆုံး Version ကိုသာ ရယူအလုပ်လုပ်ကြခြင်း ဖြစ်ပါတယ်။ Commit လုပ်လိုက်တိုင်းမှာ Update Source Code ကို Central Server ထံပေးပို့ခြင်းအားဖြင့် Version မှတ်တမ်းသစ်ကို Central Server ပေါ်မှာ တိုက်ရိုက်သိမ်းဆည်းသွားမှာ ဖြစ်ပါတယ်။

Distributed VCS တွေမှာတော့ Central Server ရှိဖို့ မလိုပါဘူး။ မည်သူမဆို မိမိတို့ Local Machine ပေါ်မှာ Repository တစ်ခုတည်ဆောက်နိုင်ပါတယ်။ အခြား Developer တွေက အဲ့ဒီ Repository ကို ရယူဖို့လိုတဲ့အခါ Repository တစ်ခုလုံးကို Clone လုပ်ယူနိုင်ပါတယ်။ ဒါကြောင့် Team မှာ ပါဝင်သူအားလုံးမှာ Repository တစ်ခုလုံးရဲ့ မူပွားတွေ ကိုယ်စီရှိနေတဲ့သဘောဖြစ်ပြီး ပြုလုပ်လိုတဲ့ ပြင်ဆင်ဖြည့်စွက်မှုတွေကို မူလ Original Repository နဲ့ ဆက်သွယ်ဖို့မလိုပဲ ကိုယ့်မူပွားပေါ်မှာ ဆက်လက် ဆောင်ရွက်ရပါတယ်။ **ပုံ (၅.၃)** မှာ လေ့လာကြည့်ပါ။



ပုံ (၅.၃) - Distributed VCS Workflow

Version တစ်ခုအဖြစ် မှတ်တမ်းတင်ဖို့ Commit လုပ်လိုက်တဲ့အခါမှာ မူလ Original Repository ကို သွားပြီး သက်ရောက် တာမျိုးမဟုတ်ပဲ ကိုယ့် Local Repository ပေါ်မှာပဲ သက်ရောက်မှာဖြစ်ပါတယ်။ Code တွေ ပေါင်းစပ်ဖို့လို အပ်လာတဲ့အခါမှာ တစ်ဦးမှာရှိနေတဲ့ Version တွေနဲ့ အခြားတစ်ဦးထံမှာရှိနေတဲ့ Version တွေကို အတွဲလိုက်ပေါင်းစပ်ယူ နိုင်ပါတယ်။

Distributed VCS တွေမှာ Version မှတ်တမ်းများ သိမ်းဆည်းခြင်း၊ Branch များခွဲယူခြင်း၊ ပြန်လည် ပေါင်းစပ်ခြင်း၊ စတဲ့ အခြေခံလုပ်ဆောင်ချက်တွေကို Network Communication မလိုပဲ Local Machine ပေါ်မှာပဲ ဆောင်ရွက်နိုင်တဲ့ အတွက် အများကြီးပိုမြန်ပါတယ်။ အားလုံးမှာ Repository မူပွားကိုယ်စီရှိနေတဲ့အတွက် Server ကွန်ပျူတာ ပျက်သွားလို့ Source Code တွေ ဆုံးရှုံးရတယ်ဆိုတာမျိုးလည်း မဖြစ်နိုင်တော့ပါဘူး။ ဒီလို အားသာချက်တွေကြောင့် အခုနောက်ပိုင်းမှာ Distributed VCS တွေကို ပိုပြီး အသုံးများလာကြပါတယ်။

5.3 – Git

Git ဟာ Linux Kernel ရဲ့ မူလဖန်တီးရှင်ဖြစ်တဲ့ Linus Torvalds ကိုယ်တိုင် (၂၀၀၅) ခုနှစ်မှာ စတင်တီထွင်ခဲ့တဲ့ Distributed VCS ဖြစ်ပါတယ်။ Open Source Software တစ်ခုဖြစ်ပြီး လက်ရှိ လူသုံးအများဆုံး VCS လည်းဖြစ်ပါတယ်။ Linus Torvalds နဲ့ Linux Kernel Developer တွေဟာ မူလက BitKeeper လို့ခေါ်တဲ့ DVCS တစ်ခုကို အသုံးပြု ပြီး Linux Kernel တစ်ခုလုံးကို ထိမ်းသိမ်းလာခဲ့ကြတာ ဖြစ်ပါတယ်။

BitKeeper ဟာ Open Source Software တစ်ခု မဟုတ်ပါဘူး။ Propitiatory Software တစ်ခုပါ။ ဒါပေမယ့် Linux Kernel Developer တွေကို ချွင်းချက် အနေနဲ့အခမဲ့အသုံးပြုခွင့် ပေးထားခဲ့ပါတယ်။ ၂၀၀၅ ရောက်တဲ့အခါမှာ ပြဿနာတစ်ချို့ကြောင့် Bitkeeper ရဲ့ မူလပိုင်ရှင်က Linux Kernel Developer တွေ အတွက် ပေးထားတဲ့ အခမဲ့အသုံးပြုခွင့်ကို ပြန်လည် ရုတ်သိမ်းလိုက်ပါတယ်။ ဒါကြောင့် Linux Kernel Source Code ကို ဆက်လက်ထိမ်းသိမ်းနိုင်ဖို့အတွက် အခြား VCS တစ်ခုနဲ့ အစားထိုးအသုံး ပြုဖို့ လိုအပ်လာပါတယ်။

Linux Kernel Development ရဲ့ဦးဆောင်သူဖြစ်တဲ့ Linus Torvalds အနေနဲ့ BitKeeper ရဲ့ Distributed သဘောသဘာဝကို နှစ်သက်သဘောကျပါတယ်။ အခြား VCS တွေကို လေ့လာကြည့်တဲ့အခါမှာတော့ အများစုက Centralize VCS တွေ ဖြစ်နေကြပြီး၊ သူလိုချင်တဲ့ အမြန်နှုန်းမရရှိပဲ အလုပ်လုပ်ပုံ နေးကွေးကြတာကို တွေ့ရပါတယ်။ ဒါကြောင့် မြန်ဆန်တိကျစွာ အလုပ်လုပ်နိုင်တဲ့ VCS တစ်ခု သူကိုယ်တိုင်ရေးဖို့ ဆုံးဖြတ်ပြီး ၂၀၀၅ ခုနှစ် ဧပြီလထဲမှာ Git ကို စတင် ဖန်တီးခဲ့ခြင်းပဲဖြစ်ပါတယ်။ ဒီလိုဖန်တီးရာမှာ အဓိကဦးတည်ချက်သုံးချက် ရှိခဲ့ပါတယ်။

1. Distributed VCS ဖြစ်ရမယ်၊
2. အမြန်နှုန်းကောင်းရမယ်၊
3. ဘယ်လိုအခြေအနေမှာပဲဖြစ်ဖြစ် Data ပျက်စီးဆုံးရှုံးခြင်း လုံးဝမရှိအောင် အာမခံနိုင်တဲ့ စနစ်တစ်ခု ဖြစ်ရမယ်

– ဆိုတဲ့ အချက်တွေပဲဖြစ်ပါတယ်။

Git Installer ကို http://git-scm.com/downloads မှာ ရယူနိုင်ပါတယ်။ Linux, Mac, Windows စတဲ့ အဓိက OS အားလုံးမှာအသုံးပြုနိုင်ပါတယ်။ Ubuntu Linux မှာဆိုရင်တော့ အောက်ပါအတိုင်း အလွယ်တစ်ကူ Install လုပ်နိုင်ပါတယ် –

```
$ sudo apt-get install git-core
```

Windows အတွက် Official Installer အစား **msysgit** လို့ခေါ်တဲ့ Package တစ်ခုကိုလည်း အစားထိုး အသုံးပြုနိုင်ပါ တယ်။ msysgit က Git သာမက Git Bash, Windows Explorer Integration စတဲ့ ဆက်စပ်ပရိဂရမ်တွေကိုပါ ထည့်သွင်းပေးတဲ့အတွက် ပိုမိုပြည့်စုံတဲ့ Installer ဖြစ်တယ်လို့ ဆိုနိုင်ပါတယ်။ msysgit ကို အောက်ပါလိပ်စာမှာ Download ရယူနိုင်ပါတယ်။

http://msysgit.github.io/

ဒီနေရာမှာ Install လုပ်ပုံအဆင့်ဆင့်ကို တစ်ဆင့်ချင်း ဖော်ပြမနေတော့ပါဘူး။ အသုံးပြုပုံကိုသာ ဆက်လက် ဖော်ပြပေး သွားမှာဖြစ်ပါတယ်။ ပြီးတော့ Git ဟာ Command Line ပရိဂရမ်တစ်ခုဖြစ်ပါတယ်။ Graphical User

Interface နဲ့ အသုံးပြုလိုတယ်ဆိုရင်လည်း UI Client တွေ ထပ်မံထည့်သွင်းအသုံးပြု နိုင်ပေမယ့် Command Line ကနေအသုံးပြုခြင်းက ပိုမိုထိရောက်ပြီးအလုပ်တွင်စေနိုင်တဲ့အတွက် Command Line ကနေ အသုံးပြုပုံကို သာ ဖော်ပြသွားမှာပါ။

> ဒီအခန်းမှာဖော်ပြသွားမယ့် Command တွေဟာ လက်တွေ့ကူးယူစမ်းသပ်လို့ရတဲ့ Command တွေ ဖြစ်ပါတယ်။ ဒါပေမယ့် Operating System အမျိုးအစား၊ Install လုပ်ထားတဲ့ Git Version, File Permission စတဲ့အချက်တွေပေါ်မူတည်ပြီး အချို့ကွဲလွဲမှုတွေ ရှိနိုင်တဲ့အတွက်၊ တိုက်ရိုက်ကူးယူ စမ်းသပ်ခြင်းမပြုပဲ အလုပ်လုပ်ပုံသဘာဝကို အဓိကထားလေ့လာသင့်ပါတယ်။

Configuration

Git ကို Install လုပ်ပြီးနောက်၊ စတင်အသုံးပြုနိုင်ဖို့အတွက် အသုံးပြုမယ့်သူရဲ့ အမည်နဲ့ Email လိပ်စာကို Configuration ဖိုင်ထဲမှာ ဖြည့်သွင်းပြီး သတ်မှတ်ပေးရပါတယ်။ Global Config နဲ့ Local Config ဆိုပြီး Configuration ဖိုင် အမျိုး အစား နှစ်မျိုးရှိပါတယ်။ စက်ထဲမှာရှိတဲ့ Git Repository အားလုံးကို သက်ရောက်စေလိုတဲ့ Setting တွေကို Global Config ထဲမှာသတ်မှတ် ပေးရပါတယ်။ သက်ဆိုင်ရာ Repository တစ်ခုကိုသာ သက်ရောက်စေလိုတဲ့ Setting တွေကို တော့ Local Config ထဲမှာ သတ်မှတ်ထားနိုင်ပါတယ်။

Configuration ဖိုင်ထဲမှာ Setting တွေ သတ်မှတ်တယ်ရတယ်ဆိုပေမယ့် ဖိုင်တွေကို လိုက်ဖွင့်ပြီး သတ်မှတ်နေစရာမလိုပါ ဘူး။ Configuration ဖိုင်တွေရဲ့ တည်နေရာဟာ အသုံးပြုတဲ့ OS နဲ့ Install လုပ်စဉ်က ရွေးချယ်ခဲ့တဲ့ အနေအထားပေါ် မူတည်ပြီး တစ်ယောက်နဲ့တစ်ယောက်မတူပဲ ကွဲပြားနိုင်ပါတယ်။ Setting တွေသတ်မှတ်ဖို့ အတွက် git config ဆိုတဲ့ Command ကိုသုံးနိုင်ပါတယ်။ အသုံးပြုမယ့်သူရဲ့ အမည်နဲ့ Email လိပ်စာကို Global Config ဖိုင်ထဲမှာ ထည့်သွင်း သွားစေဖို့ အခုလို Command ကို အသုံးပြုနိုင်ပါတယ်။

```
$ git config --global user.name "John Doe"
$ git config --global user.email "me@johndoe.com"
```

Global Config အဖြစ်သတ်မှတ်လိုတဲ့ Setting တွေကို `--global` Option နဲ့ တွဲဖက်သတ်မှတ်ရခြင်းဖြစ်ပါတယ်။ `--global` Option မပါရင်တော့ Local Config အဖြစ် သိမ်းဆည်းမှတ်တမ်းတင်သွားမှာ ဖြစ်ပါတယ်။ အမည် သတ်မှတ်ဖို့အတွက် `user.name` ကိုသုံးရပြီး Email သတ်မှတ်ဖို့အတွက် `user.email` ကို သုံးရခြင်း ဖြစ်ပါတယ်။ သတ်မှတ်ထားတဲ့ တန်ဖိုးမှန်မမှန် ပြည်လည်စီစစ်လိုရင် အခုလိုပြန်လည် စီစစ်နိုင်ပါတယ်။

```
$ git config --global user.name    # => John Doe
```

Git မှာ `user.name` နဲ့ `user.email` တို့လို Setting တွေ အများကြီးရှိပါတယ်။ အောက်ပါ Website လိပ်စာမှာ အသေးစိတ် လေ့လာနိုင်ပါတယ်။

http://git-scm.com/docs/git-config

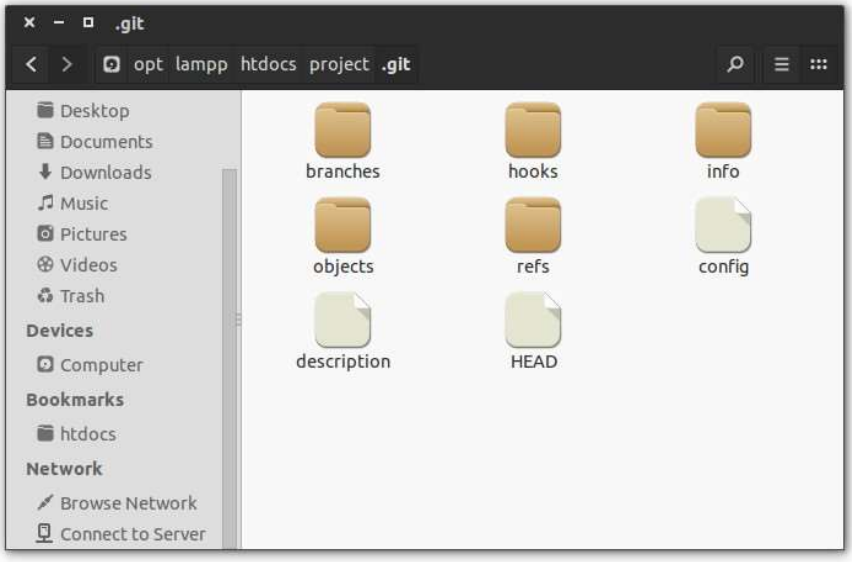
Initializing a Repository

Git ကိုအသုံးပြုပြီး Version မှတ်တမ်းစီမံနိုင်ဖို့အတွက်၊ ပထမဦးဆုံးအနေနဲ့ Source Code Directory ကို Git Repository အဖြစ် ကြေငြာသတ် မှတ်ပေးရပါတယ်။ ဒီလိုကြေငြာသတ်မှတ်ဖို့အတွက် `git init` Command ကို သုံးရပါတယ်။

```
$ git init

Initialized empty Git repository in /path/to/dir/.git/
```

နမူနာမှာပြထားသလို `git init` ကို Run လိုက်တာနဲ့ `.git` အမည်နဲ့ Empty Repository တစ်ခုကို Source Code Directory ထဲမှာ တည်ဆောက်လိုက်ကြောင်း Message ပေးလာမှာဖြစ်ပါတယ်။



ပုံ (၅.၄) - Inside .git Directory

`.git` Directory ကိုဖွင့်ကြည့်လိုက်ရင် **ပုံ (၅.၄)** မှာဖော်ပြထားသလို တွေ့ရမှာဖြစ်ပါတယ်။ Version မှတ်တမ်းတွေ Branch တွေနဲ့ အခြားလိုအပ်တဲ့ အချက်အလက်တွေကို `.git` Directory ထဲမှာ သိမ်းဆည်းသွားမှာဖြစ်ပါတယ်။

Linux အပါအဝင် Unix-like OS တွေမှာ Dot နဲ့စတဲ့ File နဲ့ Directory တွေဟာ Hidden File တွေဖြစ်တယ်ဆိုတာကို သတိပြုပါ။ File Manager နဲ့ ပြန်ကြည့်တဲ့အခါမှာပဲဖြစ်ဖြစ်၊ Terminal ထဲမှာ ကြည့်တဲ့အခါမှာပဲဖြစ်ဖြစ် Hidden File တွေကို ဖော်ပြစေမှ `.git` Directory ကို တွေ့ရမှာဖြစ်ပါတယ်။

`.git` Directory ထဲက အရေးကြီးတဲ့ နှစ်ခုကတော့ `objects` နဲ့ `hooks` တို့ဖြစ်ပါတယ်။ Git က Version မှတ်တမ်းတွေကို `objects` Directory ထဲမှာသိမ်းပါတယ်။ Encrypt လုပ်ပြီးသိမ်းမှာဖြစ်တဲ့အတွက် `objects` Directory ကိုဖွင့်ကြည့်ရင် Hash Value တွေကိုသာ တွေ့ရမှာဖြစ်ပါတယ်။ `hooks` ကိုတော့ Shell Script တွေ သိမ်း ဖို့သုံးပါတယ်။ ဥပမာ - Version မှတ်တမ်းတစ်ခု သိမ်းဆည်းဖို့အတွက် Commit လုပ်လိုက်တဲ့အခါ တွဲဖက်လုပ်ဆောင် သွားစေလိုတဲ့လုပ်ငန်းများ၊ Branch တစ်ခုနဲ့တစ်ခု Merge လုပ်လိုက်တဲ့အခါ ဆောင်ရွက်သွားစေလိုတဲ့ လုပ်ဆောင်ချက်များ စသဖြင့် Shell Script တွေရေးသားပြီး `hooks` ထဲမှာ သိမ်းဆည်းထားနိုင်ခြင်း ဖြစ်ပါတယ်။

နောက်ထပ်သတိပြုသင့်တဲ့ ဖိုင်တစ်ခုဖြစ်တဲ့ `config` ကတော့ Local Configuration File ဖြစ်ပါတယ်။ Text Editor နဲ့ ဖွင့် ကြည့်လိုက်ရင် အခုလိုတွေ့ရနိုင်ပါတယ်။

Config

```
[core]
	repositoryformatversion = 0
	filemode = true
	bare = false
	logallrefupdates = true
```

Git နဲ့အလုပ်တစ်ခုလုပ်တိုင်းမှာ ဒီ Configuration ဖိုင်ထဲမှာ ရေးသားထားတဲ့ Setting တွေကို ဦးဆုံးစီစစ်အသုံးပြုပြီး အလုပ်လုပ်ပေးသွားမှာဖြစ်ပါတယ်။ `HEAD` ဆိုတဲ့ ဖိုင်ထဲမှာတော့ လက်ရှိအလုပ်လုပ်နေတဲ့ Branch ကိုညွှန်းထားတဲ့ အညွှန်းလိပ်စာ သိမ်းဆည်းထားပါတယ်။ Version မှတ်တမ်းတစ်ခု သိမ်းဆည်းဖို့ Commit လုပ်လိုက်တိုင်းမှာ `HEAD` ဖိုင်ထဲက Branch အညွှန်းကိုကြည့်ပြီး Version မှတ်တမ်းကို သိမ်းဆည်းပေးသွားမှာဖြစ်ပါတယ်။

လက်တွေ့မှာ Version မှတ်တမ်းသိမ်းခြင်း၊ Branch တွေခွဲခြင်းစတဲ့ အခြေခံလုပ်ငန်းတွေလုပ်ဖို့အတွက် `.git` Directory ကို ပြင်ဖို့မလိုပါဘူး။ အထူးလိုအပ်ချက်ရှိတော့မှသာ ကိုယ်တိုင်ဖွင့်ပြီးစီမံဖို့လိုမှာပါ။ ဆက်လက်ဖော်ပြမယ့် Git Command တွေကို Run တဲ့အခါမှာ Source Code Directory ထဲမှာပဲ Run ရပါတယ်။ `.git` Directory ထဲမှာ Run စရာမလို ပါဘူး။

Recording and Managing Versions

`git init` နဲ့ Initialize လုပ်ထားပြီးဖြစ်တဲ့ Git Repository ထဲမှာ လိုအပ်တဲ့ Source Code ဖိုင်နဲ့ Directory တွေ တည်ဆောက်ခြင်း Code ရေးသားခြင်း၊ Compile ပြုလုပ်ခြင်းစတဲ့ လုပ်ငန်းတွေကို လုပ်ရိုးလုပ်စဉ်အတိုင်း ဆောင်ရွက်နိုင် ပါတယ်။ ရေးသားထားတဲ့ Code တွေကို Version တစ်ခုအဖြစ် မှတ်တမ်းတင်တော့မယ်ဆိုရင်တော့၊ Version မှတ်တမ်းထဲမှာ ပါဝင်စေလိုတဲ့ဖိုင်နဲ့ Directory တွေကို Index ထဲထည့်ပေးရပါတယ်။ နမူနာရစေ

ဖို့အတွက် Git Repository ထဲမှာ ဖိုင်အလွတ် တစ်ချို့ကို အခုလိုတည်ဆောက် ကြည့်နိုင်ပါတယ်။

```
$ touch index.html style.css script.js
```

`touch` Command က File တွေမရှိသေးရင် ဆောက်ပေးတယ်လို့ပဲ အလွယ်မှတ်ယူပေးပါ။ `touch` Command မသုံးချင်ရင်လည်း နှစ်သက်ရာနည်းလမ်းနဲ့ `index.html`, `style.css` နဲ့ `script.js` ဆိုတဲ့ ဖိုင်အလွတ် (၃) ခုကိုတည်ဆောက် ပေးနိုင်ပါတယ်။ ပြီးတဲ့အခါ `git status` Command ကို Run ကြည့်လိုက်ရင် အခုလို ရလဒ်ကို တွေ့ရမှာဖြစ်ပါတယ်။

```
$ git status

On branch master
Initial commit
Untracked files:
(use "git add <file>..." to include in what will be committed)

      index.html
      script.js
      style.css

nothing added to commit but untracked files present (use "git add" to track)
```

`git status` က Repository ရဲ့ လက်ရှိအခြေအနေကို လေ့လာဖို့ အသုံးပြုရခြင်းဖြစ်ပါတယ်။ ပြန်ပေးလာတဲ့ Message ရဲ့ နောက်ဆုံးလိုင်းကို လေ့လာကြည့်ရင် Version တစ်ခုအဖြစ် မှတ်တမ်းတင်စရာဘာမှ မရှိသေးတဲ့ အကြောင်း၊ ဒါပေမယ့် Index ထဲ မထည့်ရသေးတဲ့ Untracked files တွေရှိနေတဲ့အတွက် `git add` Command ကိုသုံးပြီး Track လုပ်ပေးသင့်ကြောင်း ဖော်ပြထားတာကို တွေ့ရမှာဖြစ်ပါတယ်။

`git add index.html`, `git add script.js` စသဖြင့် ဖိုင်တစ်ခုချင်းကို Index ထဲထည့်ပေးသွားလို့ ရပါတယ်။ ဖိုင်အားလုံးကို Index ထဲ ထည့်လိုရင်တော့ `git add .` လို့ ပြောလိုက်ရင်ရပါတယ်။ Dot သင်္ကေတလေးက လက်ရှိ Directory ထဲက ဖိုင်အားလုံးဆိုတဲ့ အဓိပ္ပါယ်ဖြစ်ပါတယ်။

```
$ git add .
$ git status

On branch master
Initial commit
Changes to be committed:
(use "git rm --cached <file>..." to unstage)

      new file:   index.html
      new file:   script.js
      new file:   style.css
```

`git add .` နဲ့ ဖိုင်အားလုံး Index ထဲထည့်ပြီးနောက် `git status` ကို ပြန် Run ကြည့်တဲ့အခါ၊ Version မှတ်တမ်းတင်ဖို့အတွက် ဖိုင်အသစ် (၃) ခု ရှိနေကြောင်း ပြောလာမှာဖြစ်ပါတယ်။

Index ထဲမှာ ဖိုင်အသစ် (၃) ခုရှိနေပြီး Version မှတ်တမ်းတင်ဖို့ အသင့်ဖြစ်နေပြီမို့ `git commit` Command နဲ့ မှတ်တမ်းတင်နိုင်ပါပြီ။ အဲ့ဒီလိုမှတ်တမ်းတင်ရာမှာ မှတ်ချက် (Comment) ပေးရပါတယ်။ မှတ်ချက်ကို နှစ်သက်သလိုပေး လို့ရပေမယ့် လက်ရှိမှတ်တမ်းတင်တော့မယ့် Version မှာဘာတွေတိုးလာသလဲ၊ ဘာတွေပြောင်းသွားသလဲ၊ သတိပြုသင့်တာ တွေက ဘာတွေလဲ စသဖြင့် ရှင်းပြထားတဲ့ မှတ်ချက်ဖြစ်သင့်ပါတယ်။

```
$ git commit -m "First commit"

[master (root-commit) 11bf458] First commit
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.html
 create mode 100644 script.js
 create mode 100644 style.css
```

နမူနာမှာ `git commit` Command ကို Run တဲ့အခါ `-m` Option ကိုသုံးပြီး မှတ်ချက် Comment ကို တစ်ခါတည်း တွဲဖက် ထည့်သွင်းပေးလိုက်ခြင်းဖြစ်ပါတယ်။ အဲ့ဒီလို တစ်ခါတည်းမပေးခဲ့ရင် Git က မှတ်ချက် Comment ပေးဖို့ သီးခြားထပ်မံ တောင်းဆိုလာမှာဖြစ်ပါတယ်။

ဆက်လက်ပြီး၊ နမူနာအနေနဲ့ `index.html` ကို ပြင်ပြီး `git status` ပြန် Run ကြည့်နိုင်ပါတယ်။

```
$ echo 'Hello, world!' > index.html
$ git status

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

`index.html` ဖိုင်ကို ဖွင့်ပြင်နေမယ့်အစား `echo` Command ကိုသုံးပြီး `'Hello, world!'` ဆိုတဲ့စာကို `index.html` ထဲမှာ ထည့်သွင်းလိုက်စေခြင်းဖြစ်ပါတယ်။ `echo` မသုံးချင်ရင်လည်း နှစ်သက်ရာနည်းလမ်းနဲ့ ဖိုင်ကိုဖွင့် ပြင်နိုင်ပါတယ်။

နမူနာမှာလေ့လာကြည့်ရင် `git status` က `index.html` ပြင်ထားတာကို သိရှိကြောင်း ဖော်ပြတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ပြန်ပေးလာတဲ့ Message ရဲ့ နောက်ဆုံးလိုင်းကို ကြည့်ရင် `git add` နဲ့ ပြင်ထားတဲ့ဖိုင်ကို Index ထဲ ပြန်ထည့်ပေးဖို့ ပြောထားတာကိုတွေ့ရနိုင်ပါတယ်။ ဒါမှမဟုတ် `git commit -a` ကို သုံးပြီး တစ်ခါတည်း Commit လုပ်လိုက လုပ်နိုင်ကြောင်းလည်း ပြောထားပါသေးတယ်။ Index ထဲကို တစ်ကြိမ်ထည့်

ထားဖူးတဲ့ဖိုင်ကို နောက်တစ်ခါ `git add` နဲ့ ထပ်ထည့် မနေတော့ပဲ Commit လုပ်တဲ့အခါကျမှ `add` လုပ်ပေးဖို့ `-a` Option နဲ့ တွဲပြီး အသုံးပြုနိုင် ခြင်းဖြစ်ပါတယ်။

```
$ git commit -am "Updated index.html"

[master f76103d] Updated index.html
 1 file changed, 1 insertion(+)
```

`git commit` ကို Run တဲ့အခါ `-a` Option ရော `-m` Option ပါနှစ်ခုလုံးပါအောင် `-am` လို့ တွဲပြီးအသုံးပြုလိုက် ခြင်းဖြစ်ပါတယ်။ Git က ဖိုင် (၁) ခု ပြောင်းသွားကြောင်းနဲ့ Code Line တစ်လိုင်း တိုးသွားကြောင်း ပြန်လည် ဖော်ပြလာတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။ အခု ဆိုရင် Commit နှစ်ကြိမ်ပြုလုပ်ခဲ့ပြီဖြစ်လို့ ကျွန်တော်တို့မှာ Version နှစ်ခုရရှိသွားပြီဖြစ်ပါတယ်။ `git log` Command နဲ့ Version မှတ်တမ်းကို ပြန်လည်လေ့လာနိုင်ပါ တယ်။

```
$ git log

commit f76103d35ad7b2a2641bf16d1baf6de27a5981cf
Author: Ei Maung <eimg@fairwayweb.com>
Date:   Tue Mar 17 15:42:02 2015 +0630

     Updated index.html

commit 11bf458a3a3cc45e65e5c20e16c59d92d66038e7
Author: Ei Maung <eimg@fairwayweb.com>
Date:   Tue Mar 17 12:48:56 2015 +0630

      First commit
```

`git log` Command က မှတ်တမ်းတင်ခဲ့တဲ့ Version တွေကို စာရင်းနဲ့ဖော်ပြလာခြင်းဖြစ်ပါတယ်။ Version တစ်ခု စီမှာ စာလုံး (၄၀) ပါဝင်တဲ့ Commit Hash ကိုယ်စီရှိကြပါတယ်။ Commit Hash ဆိုတာ အဲ့ဒီ Version ရဲ့ ID ပဲဖြစ်ပါ တယ်။ နောင်အဲ့ဒီ Version ကို ပြန်လိုချင်ရင် Commit Hash ကို အသုံးပြုပြီး ပြန်လည်ရယူနိုင်ပါ တယ်။ Log ထဲမှာ Commit Hash နဲ့အတူ Commit ပြုလုပ်ခဲ့သူ၊ Commit ပြုလုပ်ခဲ့တဲ့အချိန်နဲ့ Commit Comment တို့ကို စာရင်းထဲမှာ ဖော်ပြနေတာကို တွေ့ရမှာဖြစ်ပါတယ်။ `git log` ကို သပ်ရပ်သွားအောင် `--pretty` Option နဲ့ တွဲပြီး သုံးနိုင်ပါ တယ်။

```
$ git log --pretty=oneline

f76103d35ad7b2a2641bf16d1baf6de27a5981cf Updated index.html
11bf458a3a3cc45e65e5c20e16c59d92d66038e7 First commit
```

`--pretty=oneline` လို့ပြောလိုက်တဲ့အတွက် Log ကိုဖော်ပြရာမှာ စောစောကလို ရှုပ်ရှက်ခတ်နေအောင် မ ပြတော့ ပဲ Commit တစ်ခု တစ်ကြောင်းနှုန်းနဲ့ ဖော်ပြမှာဖြစ်ပါတယ်။ `online` အစား စိတ်ကြိုက် Format နဲ့

အစားထိုးသုံးနိုင်ပါတယ်။

```
$ git log --pretty=format:"%h - %an, %ar : %s"

f76103d - Ei Maung, 17 minutes ago : Updated index.html
11bf458 - Ei Maung, 3 hours ago : First commit
```

`%h` Placeholder က Short Commit Hash ဆိုတဲ့အဓိပ္ပါယ်ပါ။ Commit Hash ရဲ့ စာလုံး (၄၀) လုံးကို မဖော်ပြတော့ပဲ အတိုကောက်အနေနဲ့ ရှေ့ဆုံး (၇) လုံးကိုသာဖော်ပြတော့မှာ ဖြစ်ပါတယ်။ `%an` က Author Name, `%ar` က Author Date Relative နဲ့ `%s` ကတော့ Commit Comment Subject ဆိုတဲ့ အဓိပ္ပါယ်ပဲဖြစ်ပါတယ်။ Version တစ်ခုနဲ့တစ်ခု ကူးပြောင်းလိုတဲ့အခါ Short Commit Hash ကို သုံးပြီးတော့လည်း ကူးပြောင်းနိုင်ပါတယ်။ ဥပမာ – ပထမဆုံး Version ကိုပြန်သွားချင်ရင်၊ သူရဲ့ Short Commit Hash က `11bg458` ဖြစ်တဲ့အတွက် အခုလို ပြန်သွားနိုင်ပါတယ်။

```
$ git checkout 11bf458

Note: checking out '11bf458'.
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b new_branch_name

HEAD is now at 11bf458... First commit
```

နမူနာကိုလေ့လာကြည့်ရင် Git က လက်ရှိမှာ Commit Hash `11bf485` ကို ပြောင်းလဲအသုံးပြုနေကြောင်း၊ ဒီ Commit ကို Branch အဖြစ် ခွဲထုတ်လိုတယ်ဆိုရင်လည်း `git checkout -b` နဲ့ ခွဲထုတ်နိုင်ကြောင်း အသိပေးလာတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

တစ်ချို့အရေးပါတဲ့ Commit တွေကို နောင်ကူးပြောင်းအသုံးပြုရ လွယ်ကူစေဖို့အတွက် Tag လုပ်ထားနိုင်ပါတယ်။

```
$ git tag v1
```

လက်ရှိရောက်ရှိနေတဲ့ Commit ဖြစ်တဲ့ `11bf485` ကို `v1` လို့ Tag လုပ်ပြီး အမည်ပေးလိုက်ခြင်းဖြစ်ပါတယ်။

နောက်ဆုံး Version ကို ပြန်သွားလိုရင် `git checkout master` ကို Run ပြီး ပြန်သွားနိုင်ပါတယ်။ ဒါကြောင့် `git checkout master` နဲ့ နောက်ဆုံး Version ကိုပြန်သွားပြီး `git log` Run ကြည့်ရင် အခုလို တွေ့ရမှာဖြစ်ပါတယ်။

```
$ git checkout master
$ git log --pretty=format:'%h %s %d'

f76103d Updated index.html  (HEAD, master)
11bf458 First commit  (tag: v1)
```

`%d` Placeholder က Reference Name ဆိုတဲ့အဓိပ္ပါယ်ပါ။ နမူနာမှာကြည့်ရင် Commit `11bf458` ကို `v1` လို့ Tag လုပ်ထားကြောင်းတွေ့နိုင်ပါတယ်။ ဒါကြောင့်အဲ့ဒီ Commit ကိုသွားချင်ရင် အခုလိုအလွယ်တစ်ကူသွားလို့ ရသွားပါတယ်။

```
$ git checkout v1

Note: checking out 'v1'.
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

     git checkout -b new_branch_name

HEAD is now at 11bf458... First commit
```

ဒီနည်းနဲ့ အရေးပါတဲ့ Commit တွေကို Tag လုပ်ပြီး မှတ်သားထားနိုင်ပါတယ်။ ပေးထားတဲ့ Tag ကို ပြန်ဖျက်ချင်ရင် တော့ `git tag` Command ကို `-d` Option နဲ့ တွဲသုံးနိုင်ပါတယ်။

```
$ git checkout master
$ git tag -d v1
$ git log --pretty=format:'%h %s %d'

f76103d Updated index.html  (HEAD, master)
11bf458 First commit
```

`-d` Option က မှတ်တမ်းတင်ထားတဲ့ Commit ကိုဖျက်တာမဟုတ်ပါဘူး။ Commit ကို Reference လုပ်ထားတဲ့ Tag ကို ဖျက်ပေးခြင်းဖြစ်ပါတယ်။

Working with Branch

VCS တွေမှာ ပါဝင်လေ့ရှိတဲ့ Branch လုပ်ဆောင်ချက်ရဲ့အသုံးဝင်ပုံကို အထက်မှာဖော်ပြခဲ့ပြီးဖြစ်ပါတယ်။ Git မှာ Branch တွေစီမံဖို့အတွက် `git branch` Command ကို သုံးပါတယ်။

```
$ git branch

* master
```

လောလောဆယ် Branch တွေမရှိသေးတဲ့အတွက် `git branch` ကို Run ကြည့်တဲ့အခါ `master` တစ်ခုပဲရှိနေကြောင်း ဖော်ပြခြင်း ဖြစ်ပါတယ်။ လက်ရှိရောက်ရှိနေတဲ့ Version ကို Branch အသစ်တစ်ခုအဖြစ် အခုလို ခွဲထုတ်နိုင်ပါတယ်။

```
$ git branch dev
$ git branch

  dev
* master
```

`git branch dev` လို့ ပြောလိုက်တဲ့အချိန်မှာ `dev` ဆိုတဲ့ Branch မရှိသေးတဲ့အတွက် Git က `dev` Branch ကို အလိုအလျောက် တည်ဆောက်ပေးသွားပါတယ်။ ဒါကြောင့် နောက်တစ်ကြိမ် `git branch` Run ကြည့်တဲ့အခါမှာ `dev` နဲ့ `master` ဆိုပြီး Branch နှစ်ခု ဖြစ်သွားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ `master` ရဲ့ ရှေ့က Asterisk (*) က လက်ရှိ `master` Branch နဲ့ အလုပ်လုပ်နေကြောင်း ဖော်ပြခြင်းဖြစ်ပါတယ်။

အသစ်တည်ဆောက်ထားတဲ့ `dev` Branch ကို `git checkout` နဲ့ ကူးပြောင်းနိုင်ပါတယ်။

```
$ git checkout dev

Switched to branch 'dev'

$ git branch

* dev
  master
```

ဒီနည်းနဲ့ Branch တွေ လိုသလိုတည်ဆောက်ပြီး အပြန်အလှန် ကူးပြောင်းအလုပ်လုပ်နိုင်ပါတယ်။ Branch အသစ်တစ်ခု ဆောက်ပြီး ချက်ချင်းကူးပြောင်းသွားစေတဲ့ Short Cut အနေနဲ့ အခုလိုလည်း အသုံးပြုနိုင်ပါတယ်။

```
$ git checkout -b beta

Switched to a new branch 'beta'

$ git branch

* beta
  dev
  master
```

ဒီနည်းနဲ့ `beta` ဆိုတဲ့ Branch ကို တည်ဆောက်လိုက်ယုံမက တစ်ခါတည်း ကူးပြောင်းပေးသွားပါတယ်။ တည်ဆောက်ထားတဲ့ Branch တစ်ခုကို ပြန်လည်ပယ်ဖျက်စေလိုရင် `git branch` ကို `-d` Option နဲ့ တွဲသုံးနိုင်ပါတယ်။

```
$ git branch -d dev

Deleted branch dev (was f76103d).
```

Branch တစ်ခုကို ကူးပြောင်းပြီးနောက် ပြုလုပ်တဲ့ Commit တွေဟာ လက်ရှိရောက်ရှိနေတဲ့ Branch ကိုသာ သက် ရောက်တော့မှာ ဖြစ်ပါတယ်။

```
$ git checkout beta
$ echo 'body { margin: 0 }' > style.css
$ git commit -am "Updated style.css"

[beta d731a7f] Updated style.css
 1 file changed, 1 insertion(+)
```

```
$ git log --pretty=oneline

d731a7fcdd7dd5a478e2532e1d254a0c69184176 Updated style.css
f76103d35ad7b2a2641bf16d1baf6de27a5981cf Updated index.html
11bf458a3a3cc45e65e5c20e16c59d92d66038e7 First commit
```

`beta` Branch မှာ `style.css` ဖိုင်ကိုပြင်ပြီး Commit လုပ်လိုက်ပါတယ်။ ပြီးတော့ Log ပြန်ကြည့်တဲ့အခါ Version သုံးခု ဖြစ်သွားတာကို တွေ့ရမှာဖြစ်ပါတယ်။ `master` Branch ကို ပြန်သွားပြီး Log ပြန်ကြည့်တဲ့ အခါမှာတော့ Version နှစ်ခုသာ ရှိတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
$ git checkout master
$ git log -pretty=oneline

f76103d35ad7b2a2641bf16d1baf6de27a5981cf Updated index.html
11bf458a3a3cc45e65e5c20e16c59d92d66038e7 First commit
```

ဒီနည်းနဲ့ Source Code Repository တစ်ခုတည်းကို Branch တွေအမျိုးမျိုးခွဲပြီး စီမံဆောင်ရွက်နိုင်မှာပဲဖြစ်ပါတယ်။ Branch တစ်ခုပေါ်မှာ သိမ်းဆည်းထားတဲ့ Version တွေကိုနောက် Branch တစ်ခုနဲ့ ပေါင်းစပ်လိုရင်တော့ `git merge` ကိုသုံးပြီး ပေါင်းစပ်နိုင်ပါတယ်။

```
$ git checkout master
Switched to branch 'master'

$ git merge beta

Updating f76103d..d731a7f / Fast-forward
 style.css | 1 +
 1 file changed, 1 insertion(+)
```

`master` Branch ပေါ်မှာ `git merge beta` ကို Run လိုက်တဲ့အတွက် `beta` Branch ပေါ်က Version တွေဟာ `master` Branch နဲ့ ပေါင်းစပ်သွားမှာပဲဖြစ်ပါတယ်။ ဒါကြောင့် `master` Branch မှာ Log ကြည့်မယ်ဆိုရင် Version သုံးခုဖြစ်သွားတာကို တွေ့ရမှာပဲဖြစ်ပါတယ်။

```
$ git log -pretty=oneline

d731a7fcdd7dd5a478e2532e1d254a0c69184176 Updated style.css
f76103d35ad7b2a2641bf16d1baf6de27a5981cf Updated index.html
11bf458a3a3cc45e65e5c20e16c59d92d66038e7 First commit
```

Merge Conflicts

Branch တွေ Merge လုပ်ရာမှာ တစ်ခါတစ်ရံ Conflict တွေရှိတဲ့ပါတယ်။ Merge လုပ်လိုတဲ့ Branch နှစ်ခုလုံးက တူညီ တဲ့ဖိုင်တစ်ခုကို ပြင်ဆင်ထားမိတဲ့အခါမှာ ဖြစ်လေ့ရှိပါတယ်။ စမ်းသပ်ကြည့်နိုင်ဖို့အတွက် `master` Branch ပေါ်က `script.js` ကို အခုလို ပြင်ဆင်ပြီး Commit လုပ်လိုက်ပါမယ်။

```
$ git checkout master

Switched to branch 'master'

$ echo 'var app = {}' > script.js
$ git commit -am "Updated script.js in master"

[master 281e6f3] Updated script.js in master
 1 file changed, 1 insertion(+)
```

ပြီးတဲ့အခါ `beta` Branch ကိုကူးပြီး `script.js` ကိုပဲ ပြင်ပြီး Commit လုပ်လိုက်ပါဦးမယ်။

```
$ git checkout beta

Switched to branch 'beta'

$ echo 'var app = []' > script.js
$ git commit -am "Updated script.js in beta"

[beta 9147870] Updated script.js in beta
 1 file changed, 1 insertion(+)
```

`master` Branch ကို ပြန်သွားပြီး `beta` နဲ့ Merge လုပ်ခိုင်းတဲ့အခါ Conflict ပြဿနာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
$ git checkout master

Switched to branch 'master'

$ git merge beta

Auto-merging script.js
CONFLICT (content): Merge conflict in script.js
Automatic merge failed; fix conflicts and then commit the result.
```

`script.js` ကို `master` ရော `beta` ကပါ ပြင်ထားတဲ့အတွက် ဘာကိုအတည်ယူရမလဲ Git က မဆုံးဖြတ်နိုင်လို့ ဒီ ပြဿနာတက်ရ ခြင်းဖြစ်ပါတယ်။ `script.js` ကို ဖွင့်ကြည့်ရင် အခုလို တွေ့ရမှာပါ။

```
<<<<<<< HEAD
var app = {}
=======
var app = []
>>>>>>> beta
```

Conflict ညိနေတဲ့နေရာကို နှစ်ပိုင်းခွဲပြီး `master script.js` ထဲက Code နဲ့ `beta script.js` ထဲက Code တို့ကို ရောပေးထားပါတယ်။ ကိုယ်အတည်ပြုလိုတဲ့ အနေအထားဖြစ်အောင် Code ဖိုင်ကို စိတ်တိုင်းကျ ပြင်ပါ။ ပြီးရင် Commit လုပ်ပေးခြင်းအားဖြင့် Conflict ကို Resolve လုပ်ပြီး ဖြစ်သွားပါလိမ့်မယ်။

```
$ git commit -am "Merge with conflict resolve"

[master 08582cf] Merge with conflict resolve
```

ခုနေ Log ခေါ်ကြည့်ရင် အခုလိုတွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
$ git log --pretty=oneline

08582cf9e218e12b74ff24711692ba2efea936dd Merge with conflict resolve
91478702aebe0fe49d667316b11c6fe4bc50062f Updated script.js in beta
281e6f362f37beb9384bf0ebf15605f81f21582e Updated script.js in master
d731a7fcdd7dd5a478e2532e1d254a0c69184176 Updated style.css
f76103d35ad7b2a2641bf16d1baf6de27a5981cf Updated index.html
11bf458a3a3cc45e65e5c20e16c59d92d66038e7 First commit
```

`master` မှာ `script.js` ကိုပြင်ပြီး Commit လုပ်ခဲ့တဲ့ Version, `beta` မှာ `script.js` ကိုပြင်ပြီး Commit လုပ်ခဲ့တဲ့ Version နဲ့ Conflict ကို Resolve လုပ်ပြီး Commit လုပ်ထားတဲ့ Version အားလုံးကို မှတ်တမ်းတင်ပြီး ဖြစ် နေတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

Clone Repository

Repository တစ်ခုကို မူပွားယူဖို့အတွက် `git clone` ကို သုံးနိုင်ပါတယ်။ `git clone` နဲ့ ကိုယ့်စက်ထဲက Repository တစ်ခုကို မူပွားယူနိုင်သလို၊ Network သို့မဟုတ် အင်တာနက်ပေါ်က Repository ကိုလည်း မူပွားယူနိုင်ပါတယ်။ `git clone` နောက်ကနေ မူပွားယူလိုတဲ့ Repository ရဲ့ တည်နေရာလိပ်စာကို ညွှန်းပေးဖို့ပဲ လိုပါတယ်။ ဥပမာ –

```
$ git clone path/to/repository new_repo
$ git clone smb://192.168.1.100/repository new_repo
$ git clone ssh://server/path/to/repository new_repo
$ git clone https://github.com/user/repository new_repo
```

`git clone` နဲ့အတူ မူရင်း Repository ရဲ့လိပ်စာပဲပေးမယ်ဆိုရင် မူပွားရဲ့အမည်ကိုလည်း မူရင်း Repository အမည် အတိုင်းထားပေးသွားမှာဖြစ်ပါတယ်။ ဒါမှမဟုတ် နမူနာမှာပြထားသလို `git clone` နောက်ကနေ မူရင်း Repository လိပ်စာနဲ့အတူ မူပွားအတွက်ပေးလိုတဲ့ အမည်ကိုလည်း တွဲဖက်ပေးလိုရင် ပေးနိုင်ပါတယ်။

```
$ cd ~
$ git clone /opt/lampp/htdocs/project

Cloning into 'project'...
done.

$ ls project

index.html  script.js  style.css
```

နမူနာအနေနဲ့ `git clone` နဲ့ ကျွန်တော်တို့ စောစောကစမ်းသပ်ထားတဲ့ Repository ကို Home Directory ထဲမှာ မူပွားယူထားပါတယ်။ မူပွားရရှိလာတဲ့ project Directory ထဲက ဖိုင်စာရင်းကိုကြည့်လိုက်တဲ့အခါ ကျွန်တော်တို့ တည် ဆောက်ထားခဲ့တဲ့ `index.html`, `script.js` နဲ့ `style.css` တို့ ပါဝင်လာတာကို တွေ့ရမှာဖြစ်ပါတယ်။ ဒါ့အပြင်၊ မူပွားရရှိလာတဲ့ Repository ထဲမှာ `git log` Run ကြည့်လိုက်ရင် ကျွန်တော်တို့ မှတ်တမ်းတင်ထားခဲ့တဲ့ Version အားလုံးလည်း ပါဝင်လာတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
$ cd project
$ git log -pretty=oneline

08582cf9e218e12b74ff24711692ba2efea936dd Merge with conflict resolve
91478702aebe0fe49d667316b11c6fe4bc50062f Updated script.js in beta
281e6f362f37beb9384bf0ebf15605f81f21582e Updated script.js in master
d731a7fcdd7dd5a478e2532e1d254a0c69184176 Updated style.css
f76103d35ad7b2a2641bf16d1baf6de27a5981cf Updated index.html
11bf458a3a3cc45e65e5c20e16c59d92d66038e7 First commit
```

`git branch` Run ကြည့်ရင်တော့ `master` Branch တစ်ခုသာပါဝင်လာတာကို တွေ့ရပါလိမ့်မယ်။ ကျန် Branch တွေက မပါလာတာတော့မဟုတ်ပါဘူး၊ Reference လုပ်ယုံသာလုပ်ထားတာမို့ အခြား Branch တွေကို လိုချင်ရင် `git checkout` နဲ့ ထပ်မံရယူဖို့ လိုပါတယ်။

```
$ git branch

* master

$ git branch -a

* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/beta
  remotes/origin/master

$ git checkout beta

Branch beta set up to track remote branch beta from origin.
Switched to a new branch 'beta'

$ git branch

* beta
  master
```

နမူနာမှာ လေ့လာကြည့်ပါ။ ပထမဆုံး `git branch` Run ကြည့်တော့ `master` တစ်ခုကိုသာ တွေ့ရပါတယ်။ `git branch` ကို `-a` Option နဲ့ ထပ် Run ကြည့်တော့မှ မူလ Repository ရဲ့ Branch တွေကို ညွှန်းထားတဲ့ အညွှန်းတွေ ရှိနေတာ တွေ့ရမှာဖြစ်ပါတယ်။ ဒါကြောင့် `git checkout` နဲ့ `beta` Branch ကို ရယူပြီး `git branch` ပြန် Run ကြည့်တဲ့အခါ မှာတော့ `beta` Branch လည်း Branch စာရင်းထဲမှာ ပါဝင်လာတာကို တွေ့ရမှာပဲဖြစ်ပါတယ်။ ဒီလို မူရင်း Repository ရဲ့ Branch တွေကို ညွှန်းထားတဲ့အတွက်ကြောင့် မူရင်း Repository မှာ အပြောင်းအလဲရှိရင် ရှိတဲ့ အပြောင်းအလဲတွေကို `git pull` Command နဲ့ အလွယ်တစ်ကူ ရယူနိုင်မှာဖြစ်ပါတယ်။

```
$ cd /opt/lampp/htdocs/project
$ touch app.js
$ git add app.js
$ git commit -m "Added app.js"

[master 81f48dc] Added app.js
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 app.js

$ git log --pretty=oneline -3

81f48dc2faddcd42b0c479f53db522b64dd146ec Added app.js
08582cf9e218e12b74ff24711692ba2efea936dd Merge with conflict resolve
91478702aebe0fe49d667316b11c6fe4bc50062f Updated script.js in beta
```

နမူနာအနေနဲ့ မူရင်း Repository ထဲမှာ ဖိုင်တစ်ခုထပ်တိုးပြီး Commit လုပ်ထားပါတယ်။ Log မှာပြန်ကြည့်တဲ့အခါ ထပ်တိုးလာတဲ့ Commit ကို တွေ့ရမှာဖြစ်ပါတယ်။ `git log` မှာတွဲသုံးထားတဲ့ `-3` Option က၊ နောက်ဆုံး Commit (၃) ခုပဲကြည့်မယ်လို့ ပြောလိုက်တဲ့သဘောပါ။ `-3` အစားအခြားတန်ဖိုးတွေနဲ့ အစားထိုးအသုံးပြုနိုင်ပါတယ်။

မူပွားယူထားတဲ့ Clone Repository ထဲမှာ Log ခေါ်ကြည့်ရင်၊ မူရင်း Repository မှာပြုလုပ်ထားတဲ့ နောက်ဆုံး Commit မပါပဲ မူလ Clone လုပ်ယူစဉ်ကပါဝင်လာတဲ့ Commit တွေသာ ရှိနေတာကို တွေ့ရမှာဖြစ်ပါတယ်။

```
$ cd ~/project
$ git log --pretty=oneline -3

08582cf9e218e12b74ff24711692ba2efea936dd Merge with conflict resolve
91478702aebe0fe49d667316b11c6fe4bc50062f Updated script.js in beta
281e6f362f37beb9384bf0ebf15605f81f21582e Updated script.js in master
```

ဒါကြောင့် `git pull` နဲ့ မူရင်း Repository ထံက၊ နောက်ဆုံး Update ကို ရယူကြည့်ပါမယ်။

```
$ git pull origin master

remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
From /opt/lampp/htdocs/project
 * branch            master     -> FETCH_HEAD
   08582cf..81f48dc  master     -> origin/master
Updating 08582cf..81f48dc
Fast-forward
 app.js | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 app.js
```

`git pull` နဲ့ အတူ `origin master` ကို တွဲသုံးထားပါတယ်။ `origin` အကြောင့် ခဏနေမှပြောပါမယ်။ `master` ကတော့ Branch အမည်ဖြစ်တဲ့အတွက် `master` Branch ပေါ်က Update တွေကို ရယူမယ်လို့ ဆိုလိုက် ခြင်းဖြစ်ပါတယ်။ ဒါကြောင့် ရယူလိုတာက အခြား Branch ဆိုရင်လည်း `master` အစား၊ ရယူလိုတဲ့ Branch အမည်နဲ့ အစားထိုးအသုံးပြုနိုင်ပါတယ်။

အမှန်တော့၊ မူရင်း Repository က နောက်ဆုံး Update တွေ ရယူဖို့အတွက် `git fetch` ကို သုံးရတာပါ။ ဒါပေမယ့် `git fetch` က ရယူယုံသက်သက်ပဲ အလုပ်လုပ်ပါတယ်၊ ရရှိလာတဲ့ Update ကို လက်ရှိ Clone Repository နဲ့ပေါင်း စပ်ဖို့အတွက် `git merge` ကို ထပ်သုံးပေးရပါတယ်။ `git pull` ကတော့ Update တွေကို ရယူပေးယုံမက တစ်ခါတည်းလည်း ပေါင်းစပ်ပေးပါတယ်။ ဒါကြောင့် `git pull` ဆိုတာ `git fetch` နဲ့ `git merge` ကို ပေါင်း စပ်ထားတဲ့သဘောလို့လဲ ဆိုနိုင်ပါတယ်။

နောက်ဆုံး Update ရယူပြီး Log ပြန်ကြည့်တဲ့အခါမှာတော့ Clone Repository ထဲမှာလည်း မူရင်း Repository မှာပြုလုပ်ခဲ့တဲ့ နောက်ဆုံး Commit ပါဝင်လာတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
$ git log --pretty=oneline -3

81f48dc2faddcd42b0c479f53db522b64dd146ec Added app.js
08582cf9e218e12b74ff24711692ba2efea936dd Merge with conflict resolve
91478702aebe0fe49d667316b11c6fe4bc50062f Updated script.js in beta
```

`git pull` နဲ့အတူ တွဲသုံးခဲ့တဲ့ `origin` အကြောင်းအနည်းငယ်ပြောဖို့ လိုပါမယ်။ `origin` ဆိုတာ `git` Command မဟုတ်ပါဘူး။ အည္ဆွန်းအမည်တစ်ခုသာဖြစ်ပါတယ်။ မူရင်း Repository တည်ရှိရာ တည်နေရာကို ညွှန်းထား တဲ့ အည္ဆွန်းတစ်ခု ဖြစ်ပါတယ်။ `git clone` နဲ့ Repository ကို မူးပွားယူလိုက်စဉ်မှာ၊ Git က Clone Repository အတွက် `origin` ဆိုတဲ့ အမည်နဲ့ မူရင်း Repository တည်ရှိရာအည္ဆွန်းတစ်ခုကို အလိုအလျောက် ထည့်သွင်းပေးသွား ခြင်းဖြစ်ပါတယ်။ `git remote` Command ကို Run ကြည့်ရင် မူရင်း Repository အည္ဆွန်း စာရင်းကို တွေ့ရမှာဖြစ် ပါတယ်။

```
$ git remote

origin
```

လောလောဆယ် `origin` အမည်နဲ့ အည္ဆွန်းတစ်ခုသာ ရှိနေပါတယ်။ `origin` က ဘယ်ကိုညွှန်းထားတာလဲ သိချင်ရင် တော့ `git remote show origin` လို့ Run ကြည့်နိုင်ပါတယ်။

```
$ git remote show origin

* remote origin
  Fetch URL: /opt/lampp/htdocs/project
  Push  URL: /opt/lampp/htdocs/project
```

နမူနာအရ `origin` က `/opt/lampp/htdocs/project` Repository ကို ညွှန်းထားတာတွေ့ရမှာပါ။

Git ရဲ့ ထူးခြားချက်က Repository တစ်ခုမှာ တစ်ခုထက်ပိုတဲ့ Remote အည္ဆွန်းတွေ ရှိနိုင်ခြင်းဖြစ်ပါတယ်။ ဒီအချက်က Open Source Project တွေမှာ အတော်လေးအသုံးဝင်ပါတယ်။ ဥပမာ – Open Source Project တစ်ခုကို Team A နဲ့ Team B လို့ ခေါ်တဲ့ အဖွဲ့နှစ်ဖွဲ့က သီးခြားစီ Maintain လုပ်နေတယ်ဆိုပါစို့။ ကျွန်တော်တို့က Team A ရဲ့ Repository ကို ရော Team B ရဲ့ Repository ကိုပါ Git Remote အည္ဆွန်းများအဖြစ် ညွှန်းထားမယ်ဆိုရင်၊ Team A ရဲ့ Repository မှာရှိနေတဲ့ Update တွေရော Team B ရဲ့ Repository မှာရှိနေတဲ့ Update တွေကိုပါ တစ်နေရာထဲ ကနေ ရယူနေလို့ရနိုင်မှာပဲဖြစ်ပါတယ်။

Git Remote အည္ဆွန်းတစ်ခု ထပ်ထည့်ချင်ရင် git remote add နဲ့ ထည့်သွင်းနိုင်ပါတယ်။

```
$ git remote add teamb /var/www/project
$ git remote

teamb
origin
```

`git remote add` ကိုသုံးပြီး `teamb` ဆိုတဲ့ Remote အည္ညွန်းတစ်ခု ထပ်ထည့်လိုက်ပါတယ်။ ဒါကြောင့် `git remote` ပြန် Run ကြည့်တဲ့အခါ Remote အည္ညွန်း နှစ်ခုဖြစ်သွားတာကို တွေ့ရမှာပဲဖြစ်ပါတယ်။ `teamb` ထံက Update ကို လိုချင်ရင် –

```
$ git pull teamb master
```

– လို့ ရယူနိုင်ပြီး `origin` ထံက Update ကိုလိုချင်ရင်တော့၊ စောစောကလိုပဲ –

```
$ git pull origin master
```

– လို့ ရယူနိုင်မှာပဲဖြစ်ပါတယ်။ Remote အည္ညွန်းတစ်ခုကို ပယ်ဖျက်လိုရင် `git remote remove` ကို သုံးနိုင်ပါတယ်။

```
$ git remote remove teamb
$ git remote

origin
```

`git remote remove` နဲ့ `teamb` အည္ညွန်းကို ပယ်ဖျက်ပြီး `git remote` ပြန် Run ကြည့်တဲ့အခါ `origin` အည္ညွန်းတစ်ခုပဲကျန်တော့တာကို တွေ့ရမှာပဲဖြစ်ပါတယ်။

Centralize Repository

Git Repository တစ်ခုကို Centralize Repository တစ်ခုအနေနဲ့ တစ်ဦးထက်ပိုတဲ့ Developer တွေက ဝိုင်းဝန်းအသုံး ပြုနိုင်ပါတယ်။ အဲ့ဒီလိုအသုံးပြုနိုင်ဖို့အတွက် Working Directory မပါပဲ Version တွေ၊ Branch တွေနဲ့ အခြားလိုအပ်တဲ့ အည္ညွန်းတွေ သာပါဝင်တဲ့ Bare Repository ကို အသုံးပြုရပါတယ်။ နည်းလမ်း (၂) မျိုးနဲ့ Bare Repository တစ်ခုကို ဖန်တီးယူနိုင်ပါတယ်။ ပထမနည်းလမ်းကတော့ Directory အလွတ်တစ်ခု ထဲမှာ `git init` ကို `--bare` Option နဲ့တွဲသုံးပြီး Bare Repository အလွတ်တစ်ခုအဖြစ် ကြေငြာနိုင်ပါတယ်။

```
$ mkdir rockstar && cd rockstar
$ git init --bare

Initialized empty Git repository in /opt/lampp/htdocs/rockstar
```

`rockstar` အမည်နဲ့ Directory အလွတ်တစ်ခုတည်ဆောက်ပြီး အဲ့ဒီ Directory ထဲမှာ `git init --bare` ကို Run ခြင်းအားဖြင့် `rockstar` Directory ဟာ Git Bare Repository တစ်ခုဖြစ်သွားပါတယ်။ အဲ့ဒီ Repository ကို ဖွင့်ကြည့်လိုက်ရင် ရိုးရိုး Git Repository မှာ ပါဝင်လေ့ရှိတဲ့ `.git` Directory အတွင်းက `objects, hooks, branches, config, HEAD` စတဲ့ ဖိုင်နဲ့ Directory တွေကို တွေ့ရမှာပဲဖြစ်ပါတယ်။ တနည်းအားဖြင့် `rockstar` ဟာ Source Code ဖိုင်တွေသိမ်းဆည်းဖို့ မဟုတ်ပဲ၊ Version မှတ်တမ်းတွေကိုသာ သိမ်းဆည်းဖို့ ရည်ရွယ် တည်ဆောက်လိုက်တဲ့ Repository တစ်ခု ဖြစ်သွားခြင်း ဖြစ်ပါတယ်။ ဒါကြောင့် Source Code ဖိုင်တွေကို သူ့ထဲမှာ တိုက်ရိုက် ရေးသားတည်ဆောက်ခြင်း မလုပ်ရတော့ပါဘူး။ သူ့ကို Version မှတ်တမ်း သိမ်းဆည်းပေးတဲ့ Centralize Repository အဖြစ်သာ မှတ်ယူရတော့မှာပါ။

Bare Repository တစ်ခုတည်ဆောက်နည်း နောက်တစ်နည်းကတော့ Clone လုပ်ယူစဉ်မှာ `--bare` Option ကို အသုံးပြုခြင်းပဲ ဖြစ်ပါတယ်။

```
$ cd ~
$ git clone --bare /opt/lampp/htdocs/project

Cloning into bare repository 'project.git'...
done.
```

ဒီနည်းနဲ့ မူလ Source Code တွေနဲ့ Version မှတ်တမ်းတွေ ရှိနေပြီးသား Repository တစ်ခုကို Bare Repository တစ်ခုအဖြစ် မူပွားယူနိုင်ခြင်းဖြစ်ပါတယ်။ နမူနာမှာလေ့လာကြည့်ရင် Git က မူလ Repository ဖြစ်တဲ့ project ကို `project.git` အမည် ရှိတဲ့ Bare Repository အဖြစ် Clone လုပ်ပေးသွားခြင်းဖြစ်တာကို တွေ့ရနိုင်ပါတယ်။ Git Repository Directory တွေကို အမည်ပေးတဲ့အခါ နောက်ဆုံးက `.get` ထည့်ပြီး ပေးကြလေ့ရှိပါတယ်။ မဖြစ်မနေပေးဖို့ လိုတာမျိုးမဟုတ်ပေမယ့် အမည်ကို ကြည့်လိုက်ယုံနဲ့ Git Repository ဖြစ်ကြောင်း သိသာစေနိုင်လို့ Git Repository အဖြစ်သုံးမယ့် Directory တွေကို အမည်ပေးတဲ့အခါ အသုံးပြုသင့်တဲ့ နည်းလမ်းတစ်ခုပဲဖြစ်ပါတယ်။

တည်ဆောက်ထားတဲ့ Bare Repository ကို Centralize Repository အဖြစ် Project မှာပါဝင်သူအားလုံး Access လုပ်နိုင်မယ့် တစ်နေရာမှာ ထားဖို့လိုပါတယ်။ Developer တစ်ဦးရဲ့စက်ထဲမှာထားပြီး Network Share အဖြစ် Share ပေးလိုက်ရင်လည်း ရပါတယ်။ ဒါမှမဟုတ် ပိုပြီးစနစ်ကျစေချင်ရင် သီးခြား Server ကွန်ပျူတာတစ်လုံးမှာထားပြီး၊ SSH သို့မဟုတ် HTTP တို့ကနေ တစ်ဆင့် ရယူနိုင်အောင်လည်း ထားနိုင်ပါတယ်။ ဒီနေရာမှာတော့ SSH Server နဲ့ HTTP Server များ Setup လုပ်ပုံတို့ကို ထည့်သွင်းမဖော်ပြနိုင်တော့ပါဘူး။ သဘောသဘာဝပိုင်းကိုသာ ဆက်လက်ဖော်ပြပေးသွား ပါမယ်။

Project မှာပါဝင်သူတစ်ဦးအနေနဲ့ Code တွေ စတင်ရေးသားနိုင်ဖို့အတွက် ပထမဦးဆုံးအနေနဲ့ Centralize Repository အဖြစ်သတ်မှတ်ထားတဲ့ Bare Repository ကို Clone လုပ်ယူဖို့လိုပါတယ်။ Clone လုပ်ပုံလုပ်နည်းက ရိုးရိုး Repository ကို Clone လုပ်ပုံနဲ့အတူတူပဲ ဖြစ်ပါတယ်။

```
$ git clone /opt/lampp/htdocs/rockstar

Cloning into 'rockstar'...
warning: You appear to have cloned an empty repository.
done.
```

ဘာမှမရှိသေးတဲ့ Repository အလွတ်ဖြစ်နေတယ်ဆိုရင် Git က နမူနာမှာပြထားသလို Waring တစ်ခုပေးနိုင်ပါတယ်။ Clone ရယူထားတဲ့ Local Repository ထဲမှာ Source Code ဖိုင်နဲ့ Directory တွေတည်ဆောက်ခြင်း၊ Code များရေးသားခြင်း နဲ့ Version မှတ်တမ်းများ Commit လုပ်ခြင်းတို့ကို လုပ်ရိုးလုပ်စဉ်အတိုင်း ဆက်လက်ဆောင်ရွက် နိုင်ပါတယ်။

Clone မလုပ်ပဲ Git Repository အလွတ်တစ်ခုတည်ဆောက်ပြီး `git remote add` နဲ့ Centralize Repository ကို ညွှန်းပေးလို့လည်းရပါတယ်။ ဥပမာ -

```
$ git init

Initialized empty Git repository in /home/eimg/rockstar/.git/

$ git remote add origin /opt/lampp/htdocs/rockstar
```

ကိုယ်ရဲ့ Local Repository ထဲမှာ မှတ်တမ်းတင်ထားတဲ့ Version မှတ်တမ်းကို Centralize Repository ထံ ပေးပို့သိမ်းဆည်းလိုတဲ့အခါ `git push` ကို သုံးရပါတယ်။

```
$ git push origin master

Counting objects: 3, done.
Writing objects: 100% (3/3), 228 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /opt/lampp/htdocs/rockstar
* [new branch]      master -> master
```

စောစောကကြည့်ခဲ့တဲ့ `git pull` မှာလိုပဲ `origin` ဆိုတဲ့ Remote အညွှန်းနဲ့ `master` ဆိုတဲ့ ပေးပို့လိုတဲ့ Branch တို့ကို `git push` နဲ့အတူ တွဲဖက်ပေးရခြင်း ဖြစ်ပါတယ်။ ကိုယ်က Version မှတ်တမ်းတွေကို Centralize Repository ထံပေးပို့နေသလို အခြား Developer များကလည်း ပေးပို့နေမှာမို့ Centralize Repository ထံကနေ နောက်ဆုံး Update ကိုလည်း ပြန်လည်ရယူဖို့ လိုပါသေးတယ်။ Centralize Repository မှာ Update ရှိနေတာကို မရယူပဲနဲ့လည်း ကိုယ့်ရဲ့ Version တွေကို ပေးပို့လို့ရမှာ မဟုတ်ပါဘူး။

```
$ git push origin master

To /opt/lampp/htdocs/rockstar
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to '/opt/lampp/htdocs/rockstar'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

နမူနာကိုလေ့လာကြည့်ရင် Centralize Repository က ပေးပို့လာတဲ့ Version တွေကို လက်ခံဖို့ ငြင်းပယ်နေတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ Centralize Repository ထဲမှာ Update တွေရှိနေတဲ့အတွက် အဲ့ဒီ Update တွေကို `git pull` နဲ့ အရင်ရယူပြီးမှ ပြန်လည်ပေးပို့ဖို့လည်း ပြောထားပါသေးတယ်။

```
$ git pull origin master

remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
Unpacking objects: 100% (3/3), done.
From /opt/lampp/htdocs/rockstar
 * branch            master     -> FETCH_HEAD
   6a05a77..b7ab351  master     -> origin/master
Merge made by the 'recursive' strategy.
 LICENSE | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 LICENSE
```

`git pull` နဲ့ Centralize Repository က နောက်ဆုံး Update ကိုရယူတဲ့အခါ ကိုယ့် Local Version တွေနဲ့ Centralize Repository ကပြန်ပေးလာတဲ့ Version တွေကို Git က Merge လုပ်ပေးသွားမှာဖြစ်ပါတယ်။ ဒီနေရာမှာ တစ်ခါတစ်ရံ Conflict တွေရှိတက်ပါတယ်။ Branch တွေ Merge လုပ်တုန်းကလိုပဲ Local မှာ ကိုယ်ပြင်ထားတဲ့ ဖိုင်နဲ့ Centralize Repository က ပြန်ပေးလာတဲ့ နောက်ဆုံး Version ထဲက ပြင်ထားတဲ့ ဖိုင်ချင်းတိုက်နေရင် Merge Conflict ဖြစ်တက်ခြင်း ဖြစ်ပါတယ်။ အဲ့ဒီလို Conflict တွေရှိလာရင်တော့ Conflict ရှိနေတဲ့ဖိုင်ကို လိုသလိုဖွင့်ပြင်ပြီး Commit ပြန်လုပ်ပေးဖို့လိုမှာဖြစ်ပါတယ်။

နောက်ဆုံး Update ရယူပြီးနောက်ကိုယ့်ရဲ့ Local Version Update ကို Centralize Repository ထံ ပြန်လည်ပေးပို့ လိုက ပေးပို့နိုင်ပြီးဖြစ်ပါတယ်။

```
$ git push origin master
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 604 bytes | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To /opt/lampp/htdocs/rockstar
   b7ab351..72044c5  master -> master
```

ဒီနေရာမှာ တစ်ခုသတိပြုရမှာက၊ Bare Repository တစ်ခုထံကိုသာ Push လုပ်လို့ရပါတယ်။ ရိုးရိုး Repository တစ်ခု ကနေ Clone လုပ်ထားရင်တော့ မူရင်း Repository ထံက Update ကို Pull လုပ်ယူလို့ရပေမယ့်၊ ကိုယ့် Update ကိုတော့ ပြန်လည်ပေးပို့လို့ ရမှာမဟုတ်ပါဘူး။ ဒါကြောင့် Centralize Repository အဖြစ်ထားရှိလိုတဲ့ အခါတွေမှာ Bare Repository တွေကိုသာ Centralize Repository အဖြစ် ထားရှိအသုံးပြုရခြင်းဖြစ်ပါတယ်။

Other

Git မှာ အခြားအသုံးဝင်တဲ့ အခြေခံလုပ်ဆောင်ချက် တစ်ချို့ ရှိပါသေးတယ်။ Commit ပြုလုပ်လိုက်ပြီးမှာ အဲ့ဒီ Commit ကို ပြန်လည်ပြင်ဆင်လိုတယ်ဆိုရင် `--amend` Option ကို သုံးနိုင်ပါတယ်။

```
$ git commit --amend -am "Commit comment"
```

နမူနာမှာပေးထားသလို Commit ပြုလုပ်စဉ်မှာ `--amend` Option ကိုထည့်သွင်းအသုံးပြုခဲ့ရင် Commit အသစ်အနေနဲ့ မှတ်တမ်းမတင်တော့ပဲ နောက်ဆုံးပြုလုပ်ခဲ့တဲ့ Commit အစား အစားထိုးမှတ်တမ်းတင်သွားမှာဖြစ်ပါတယ်။

အရင်ပြုလုပ်ခဲ့တဲ့ Commit တွေကို အပြီးဖျက်ပြစ်လိုရင်တော့ `git reset` ကို သုံးနိုင်ပါတယ်။ ဖျက်လိုတဲ့ Commit ရဲ့ Hash ID ကိုသုံးရမှာဖြစ်လို့ Log ထဲမှာ Commit ID ကိုအရင်ကြည့်ဖို့တော့လိုပါတယ်။

```
$ git log --pretty=format:"%h %s"

72044c5 Merge branch 'master' of /opt/lampp/htdocs/rockstar
9e4b188 Added index.html
b7ab351 Added LICENSE
6a05a77 Added README
```

```
$ git reset --hard b7ab351

HEAD is now at b7ab351 Added LICENSE

$ git log --pretty=format:"%h %s"

b7ab351 Added LICENSE
6a05a77 Added README
```

နမူနာကိုလေ့လာကြည့်ရင် မူလက Commit (၅) ခုရှိခဲ့ပေမယ့် `git reset --hard b7ab351` လို့ပြောလိုက်တဲ့ အချိန်မှာ နောက်ဆုံး Version အဖြစ် `b7ab351` ကို သတ်မှတ်လိုက်တဲ့အတွက် `b7ab351` နောက်ပိုင်း Commit တွေ အားလုံးပျက်သွားမှာဖြစ်ပါတယ်။ ဒီနည်းလမ်းဟာ Source Code တွေ Loss ဖြစ်နိုင်လို့ ထူးခြားတဲ့လိုအပ်ချက်မရှိရင် မသုံးသင့်တဲ့နည်းလမ်းတစ်ခုဖြစ်ပါတယ်။

နောက်ထပ်သတိပြုသင့်တဲ့အချက်တစ်ချက် ရှိပါသေးတယ်။ Windows, Mac, Linux စတဲ့ Operating System တွေမှာ သုံးကြတဲ့ Line Ending နည်းစနစ်တွေ ကွဲပြားတက်ပါတယ်။ တစ်ချို့က Carriage Return (CR) သင်္ကေတကို Line Ending အဖြစ် သုံးကြပါတယ်။ တစ်ချို့က Line Feed (LF) သင်္ကေတကိုသုံးပါတယ်။ တစ်ချို့ကတော့ နှစ်ခုလုံး ပေါင်းပြီး (CRLF) ဆိုပြီး သုံးပါတယ်။ Developer တစ်ယောက်က Linux နဲ့ရေးထားတဲ့ Source Code ကို နောက် Developer တစ်ယောက်က Windows မှာ ဖွင့်လိုက်တဲ့အခါ Line Ending ပြောင်းသွားနိုင်ပါတယ်။ အဲ့ဒီလို ပြောင်းသွားတဲ့အခါ Git က Source Code ကို ပြင်လိုက်တယ်လို့ ယူဆလိုက်မှာပါ။ တစ်ကယ်တမ်းက ပြင်လိုက်တာမဟုတ်ပဲ OS မတူလို့ Line Ending ပြောင်းသွားတာပါ။ ဒီပြဿနာမျိုး မတက်စေဖို့ အတွက် Windows အသုံးပြုသူတွေဟာ ဒီ Setting ကို သတ်မှတ်ထားသင့်ပါတယ်။

```
$ git config --global core.autocrlf true
$ git config --global core.safecrlf true
```

Mac သို့မဟုတ် Linux အသုံးပြုသူတွေကတော့ ဒီလို သတ်မှတ်ထားသင့်ပါတယ်။

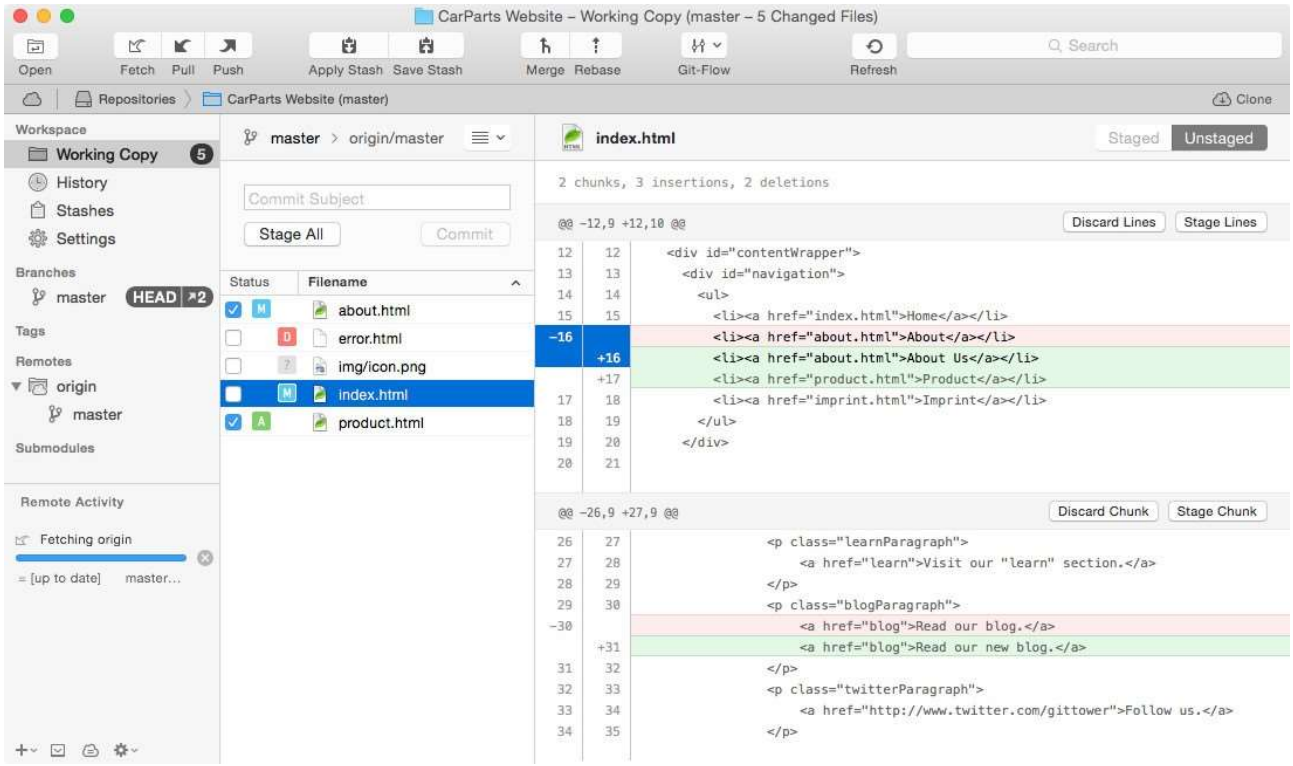
```
$ git config --global core.autocrlf input
$ git config --global core.safecrlf true
```

ဒီတော့မှ Git က Line Ending ကို သင့်တော်အောင် အလိုအလျောက် Convert လုပ်ပြီး အလုပ်လုပ်ပေးသွားမှာ ဖြစ်လို့ OS မတူလို့ Line Ending ပြောင်းပြီးဖြစ်ရတဲ့ ပြဿနာမဖြစ်တော့မှာပဲဖြစ်ပါတယ်။ Windows Setting မှာ `autocrlf true` လို့ ပြောထားတဲ့အတွက် Commit လုပ်ချိန်မှာ Line Ending ကို LF ပြောင်းပြီးမှ Commit လုပ်ပေးသွားမှာဖြစ်ပါတယ်။ Checkout လုပ်ချိန်မှာတော့ Windows Line Ending ဖြစ်တဲ့ CRLF ပြောင်းပြီး Checkout လုပ်ပေးသွား မှာပဲဖြစ်ပါတယ်။ Mac နဲ့ Linux အတွက်တော့ `autocrlf input` လို့ ပြောထားတဲ့အတွက် Checkout မှာ Line Ending Conversion ကို လုပ်မနေတော့ပဲ Commit မှာပဲ Line Ending Conversion ကို လုပ် ပေးသွားမှာဖြစ်ပါတယ်။ `safecrlf true` Setting ကတော့ Line Ending Conversion လုပ်လိုက်တဲ့အတွက် Source Code ရဲ့ Integrity ကို ထိခိုက်မသွားအောင်စစ်ဆေးဖို့ သတ်မှတ်ထားခြင်းဖြစ်ပါတယ်။ `safecrlf` လို့ ပြောထားတဲ့အတွက် Source Code ကို ထိခိုက်နိုင်စရာ ရှိတယ်ဆိုရင် Convert မလုပ်တော့ပဲ Commit တွေကို ငြင်းပယ် သွားမှာ ဖြစ်ပါတယ်။

နောက်ထပ်သတိပြုစရာတစ်ချက်ကတော့ Indention ဖြစ်ပါတယ်။ Code တွေရေးသားတဲ့အခါ Indent တွေ မဖြစ်မနေ ပါဝင်ကြပါတယ်။ တစ်ချို့က Tab ကို Indent လုပ်ဖို့သုံးပြီး တစ်ချို့ကတော့ Space ကို Indent လုပ်ဖို့ သုံးပါတယ်။ ဘာသုံးသုံး တစ်မျိုးတည်း သုံးမယ်ဆိုရင်ရပါတယ်။ Tab နဲ့ Space ကို ရောသုံးတဲ့အခါမှာတော့ ပြဿနာရှိတက်ပါတယ်။ Developer တစ်ယောက် က Indent အတွက် Space သုံးထားတယ်။ နောက် Developer တစ်ယောက်က Code ကို ပြင်တာမဟုတ်ပေမယ့် တစ်နေရာမှာ Indent Space အစား Tab ထည့်လိုက်မိတယ်ဆိုရင် Space နဲ့ Indent တွေရော နေပြီး မလိုလားအပ်တဲ့ ပြဿနာတစ်ချို့ ရှိတက်ပါတယ်။ Git ကတော့ Default အနေနဲ့ Tab နဲ့ Space တွေရောပြီးသုံး ထားရင် Commit တွေကို လက်မခံပဲ ငြင်းပယ်သွားမှာ ဖြစ်ပါတယ်။ ဒီပြဿနာကိုရှောင်ရှားဖို့အတွင် Git Setting ကို ပြင်မနေပဲ ရေးသားသူတွေကြားထဲမှာ Coding Standard တစ်ခု ထားပြီးတော့ ညှိနှိုင်းကြဖို့ လိုအပ်ပါတယ်။

Git ကို Command Line ကနေမသုံးပဲ User Interface နဲ့ သုံးချင်သူများအနေနဲ့ TortoiseGit, Giggle, Tower စတဲ့ UI Software တွေကို အသုံးပြုနိုင်ပါတယ်။ တစ်ချို့ အခမဲ့ရပြီး တစ်ချို့ လိုင်စင်ဝယ်သုံးရပါတယ်။ အောက်ပါလိပ်စာမှာ အသုံးများတဲ့ Git GUI Software စာရင်းကို လေ့လာနိုင်ပါတယ်။

http://git-scm.com/downloads/guis



ပုံ (၅.၅) - Git Tower – Git GUI Software on Mac

နောက်ဆုံးတစ်ချက်အနေနဲ့၊ Git Repository တွေကို အဖွဲ့လိုက်ပူးပေါင်းဆောင်ရွက်နိုင်ဖို့ Centralize Server တစ်ခုနဲ့ Setup လုပ်ရာမှာ Github (github.com) နဲ့ BitBucket (bitbucket.org) တို့လို Source Code Hosting Service တွေကိုလည်း လေ့လာအသုံးပြုသင့်ကြောင်း အကြံပြုလိုပါတယ်။ ဒီ Service တွေရဲ့အကူအညီနဲ့ Centralize Server တစ်ခုကို ကိုယ်တိုင် Setup လုပ်နေစရာမလိုပဲ အလွယ်တစ်ကူ ရရှိနိုင်ပါတယ်။

Software Versioning

ဆက်လက်ဖော်ပြချင်တာကတော့ Software တစ်ခုကို Version နံပါတ်တွေ ပေးပုံပေးနည်းဖြစ်ပါတယ်။ အခြေခံအားဖြင့် Commit တစ်ခုဟာ Version တစ်ခုဖြစ်တယ်လို့ အကြမ်းဖျင်းအားဖြင့် ပြောနိုင်ပါတယ်။ ဒါပေမယ့် မှတ်သားရခက်တဲ့ Commit Hash တွေကို Version နံပါတ်အဖြစ် သုံးလို့တော့ အဆင်မပြေပါဘူး။ အရေးပါတဲ့ Commit တွေကို နားလည် မှတ်သားရ လွယ်တဲ့ Version နံပါတ်တွေ သတ်မှတ်ထားပေးဖို့လိုပါတယ်။ Git ရဲ့ Tag လုပ်ဆောင်ချက်ကို ဒီနေရာမှာ အသုံးချနိုင်ပါတယ်။

Github ကိုပူးတွဲတည် ထောင်ခဲ့သူတစ်ဦးဖြစ်တဲ့ Tom Preston-Werner ရေးသားတင်ပြထားတဲ့ Semantic

Versioning ဆိုတဲ့ Version နံပါတ်သတ် မှတ်ရာမှာ လိုက်နာသင့်တဲ့ နည်းလမ်းများဆိုတာ ရှိပါတယ်။ သတ်မှတ်ချက်အပြည့်အစုံပါဝင် တဲ့ Version နံပါတ်တစ်ခုဟာ **ပုံ (၅.၆)** မှာဖော်ပြထားသလို ပုံစံဖြစ်မှာပါ။

ပုံ (၅.၆) - Semantic Version Number

Software တစ်ခု ပထမဆုံးစတင်စမ်းသပ်အသုံးပြုလို့ရတဲ့ အဆင့်ကို Version 0.1.0 လို့သတ်မှတ်ရမှာ ဖြစ်ပါတယ်။ အဲဒီအဆင့်မှာ Software က စပြီးစမ်းသုံးလို့ရနေပါပြီ။ ဒါပေမယ့် သတ်မှတ်လုပ်ဆောင်ချက်တွေတော့ စုံလင်အောင် မပါဝင် သေးပါဘူး။ အဲ့ဒီ အဆင့်မှာပဲ အမှားတစ်ချို့ စတွေ့လို့ ပြင်ဆင်လိုက်မယ်ဆိုရင် Version 0.1.1, 0.1.2, 0.1.3 စသဖြင့် ပြင်ဆင်ချက်အလိုက် နောက်ဆုံးက PATCH နံပါတ်လည်း လိုက်တိုးလာမှာဖြစ်ပါတယ်။

နောက်ထပ်ဖြည့်စွက်လုပ်ဆောင်ချက်တစ်ချို့ ဖြည့်စွက်လိုက်လို့ ပိုပြည့်စုံလာပြီဆိုရင်တော့ Version 0.2.0 အဖြစ်သတ် မှတ်ရမှာဖြစ်ပါတယ်။ 0.2.0 အဆင့်မှာ ပြုလုပ်ခဲ့တဲ့ အမှားပြင်ဆင်ချက်တွေကိုတော့ 0.2.1, 0.2.2, 0.2.3 စသဖြင့်အဆင့်လိုက်ပေးသွားရမှာဖြစ်ပါတယ်။ ဒီနည်းနဲ့ 0.3.0, 0.3.2, 0.3.2, 0.4.0, 0.4.1 စသဖြင့် လုပ်ဆောင်ချက်တိုးလာတိုင်း MINOR Version နံပါတ်တိုးလာသလို့ MINOR Version တစ်ခုချင်းစီမှာ ပြင်ဆင်မှုတွေ လုပ်လိုက်တိုင်း PATCH Version နံပါတ်တွေလည်း တိုးလာမှာပဲဖြစ်ပါတယ်။

ဒီနည်းနဲ့ နောက်ဆုံး အများအသုံးပြုနိုင်ဖို့ ကြေငြာနိုင်လောက်အောင် ပြည့်စုံလာပြီဆိုရင်တော့ MAJOR Version နံပါတ် တိုးသွားပြီး 1.0.0 ဖြစ်သွားမှာပဲဖြစ်ပါတယ်။ ဒါပေမယ့် အများကိုကြေငြာတော့မှာမို့ ပြည့်စုံပြီးအမှားနည်းဖို့လိုပါတယ်။ အသေအချာမစမ်းသပ်ရသေးတဲ့ အခြေအနေမှာ 1.0.0-pre-alpha လို့ သတ်မှတ်ထားသင့်ပါတယ်။ စမ်းသပ်မှုတွေ စတင်ပြီဆိုရင်တော့ 1.0.0-alpha ကိုယ်တိုင်စမ်းသပ်ယုံမက ပြင်ပ User တွေကိုလည်း စတင်စမ်းသပ်ခွင့်ပြုပြီဆိုရင် 1.0.0-beta စသဖြင့် သတ်မှတ်သွားရမှာဖြစ်ပါတယ်။ အဲ့ဒီ pre-alpha, alpha နဲ့ beta အဆင့်တွေမှာ အမှားပြင်ဆင်ချက် တွေ ရှိခဲ့တယ်ဆိုရင် PRE-RELEASE PATCH နံပါတ်ကို တိုးသွားနိုင်ပါတယ်။ 1.0.0-beta.1, 1.0.0-beta.2 စသဖြင့် ပေးသွားနိုင်ပါတယ်။

စိတ်ကျေနပ်လောက်အောင် စမ်းသပ်ပြုပြင်မှုတွေဆောင်ရွက်ပြီးလို့ အသင့်ဖြစ်နေပြီဆိုရင်တော့ 1.0.0-rc လို့ သတ်မှတ်နိုင်ပါတယ်။ rc က Release Candidate ဆိုတဲ့အဓိပ္ပါယ်ပါ။ ဒီအတိုင်း Release လုပ်တော့မယ်လို့ ဆုံးဖြတ်လိုက်တဲ့အဆင့်ဖြစ် ပါတယ်။ 1.0.0-rc အဆင့်မှာ ပြဿနာတစ်စုံတစ်ရာမရှိတော့ဘူးဆိုရင်တော့ PRE-RELEASE ကိုဖြုတ်လိုက်ပြီး Version 1.0.0 အနေနဲ့ အများ အသုံးပြုနိုင်အောင် ကြေငြာနိုင်ပြီပဲဖြစ်ပါတယ်။

1.0.0 အဖြစ် ကြေငြာပြီးနောက် အသေးစားပြင်ဆင်ချက်တွေကို 1.0.1, 1.0.2 စသဖြင့် ဆက်လက် Release လုပ်သွားလို့ ရပါတယ်။ အသေးစားဖြည့်စွက်ချက်တွေပြုလုပ်ခဲ့ရင်တော့ 1.1.0, 1.2.0 စသဖြင့် ဆက်လက် Release လုပ်သွားနိုင်ပါတယ်။ အဲ့ဒီအဆင့်တွေမှာလည်း 1.1.0-beta, 1.2.0-rc စသဖြင့် PRE RELEASE သတ်မှတ်ချက်တွေ လိုအပ်ရင်တွဲသုံးနိုင် ပါသေးတယ်။

ပြင်ဆင်ဖြည့်စွက်ချက်တွေ တစ်ဖြည်းဖြည်းများလာပြီး၊ သိသာမြင်သာတဲ့ နောက်အဆင့်တစ်ခုကို ရောက်လာပြီဆိုရင်၊ (သို့မဟုတ်) Software ကို နည်းစနစ်သစ်၊ ပုံစံသစ်တွေနဲ့ ပြန်လည်ဆန်းသစ်တော့မယ်ဆိုရင်တော့ 2.0.0 ဆိုပြီး MAJOR Version နံပါတ်သစ်တစ်ခုနဲ့ ဆက်လက်ဆောင်ရွက်ရမှာဖြစ်ပါတယ်။

Commit Hash (သို့မဟုတ်) Software ကို Build လုပ်လိုက်ရာမှာ ရရှိလာတဲ့ Unique ID (သို့မဟုတ်) Release လုပ်ခဲ့တဲ့ရက်စွဲ စတဲ့အချက်အလက်တွေကို Version နံပါတ်နဲ့တွဲဖက်မယ်ဆိုရင် နောက်ဆုံးကနေ + သင်္ကေတနဲ့ တွဲဖက်နိုင် ပါတယ်။ ဒါပေမယ့် အချက်အလက်အနေနဲ့သာ ထည့်သွင်းသင့်ပြီး အဲ့ဒီအချက်အလက်ကို မှီခိုအလုပ်လုပ်တာမျိုးတော့ မလုပ်ရပါဘူး။ ဥပမာ – Version 2.5.1-rc.2+72044c5, 1.4.3+20150318 စသဖြင့် တွဲဖက်ပေးနိုင်ပါတယ်။

Software တိုင်းက ဒီနည်းတစ်မျိုးတည်းကို သုံးနေတာတော့ မဟုတ်ပါဘူး။ တစ်ချို့လည်း Version ကို နံပါတ်တင်မက နာမည်နဲ့လည်း ပေးတက်ကြပါတယ်။ ဥပမာ - Flash MX, Photoshop CS, Windows XP။ တစ်ချို့လည်း ခုနှစ်နဲ့ ပေးတက်ကြပါတယ်။ ဥပမာ – Windows 2000, Ubuntu 14.04။ တစ်ချို့လည်း Internal Version နံပါတ်နဲ့ အများ ကို ကြေငြာတဲ့နံပါတ်ဆိုပြီး နှစ်မျိုးထားတက်ကြပါတယ်။ ဥပမာ – Java7 ရဲ့ တစ်ကယ့် Internal Version နံပါတ်က Java 1.7 ဖြစ်ပါတယ်။ လက်တွေ့မှာ Version နံပါတ်တွေကို ကိုယ့်နည်းကိုယ့်ဟန်နဲ့ အမျိုးမျိုးပေးတက်ကြပါတယ်။ ဒါကြောင့် အခုဖော်ပြခဲ့တဲ့နည်းအတိုင်းအမြဲပေးရမယ်လို့တော့ မဆိုလိုပါဘူး။ ဒါပေမယ့် ဒီနည်းစနစ်ဟာ အများစံထားသုံး နေတဲ့ နည်းစနစ်တစ်ခုဖြစ်လို့ လိုက်နာအသုံးပြုသင့်တဲ့နည်းစနစ်တစ်ခုပဲဖြစ်ပါတယ်။

Conclusion

Software Project တစ်ခုကို တစ်ဦးတည်းရေးသားသည်ဖြစ်စေ၊ Team နဲ့ ပူးပေါင်းရေးသားသည်ဖြစ်စေ Version Control System တစ်ခုကိုတော့ မဖြစ်မနေ အသုံးပြုသင့်ပါတယ်။ တစ်ချို့လည်း Centralize Code Base လိုအပ်ပြီး အဖွဲ့လိုက်ရေးသားရတဲ့ Project တွေမှသာ VCS တွေလိုတယ်လို့ အမှတ်မှားကြပါတယ်။ လက်တွေ့မှာ Version Control System တွေက Centralize Code Base တစ်ခုရရှိအောင် အကူအညီပေးယုံမှ မက၊ အဆင့်လိုက် မှတ်တမ်းတင်ပေးတဲ့ အတွက် Code Loss မဖြစ်အောင် ကာကွယ်ပေးနိုင်ခြင်းနဲ့၊ Branch လုပ်ဆောင်ချက်နဲ့အတူ Code Base တစ်ခုတည်းကို သုံးပြီး မတူကွဲပြားတဲ့ Version တွေကို ရေးသားစီမံနိုင်အောင်လည်း ကူညီပေးနိုင်လို့ တစ်ဦးတည်းရေးသားတဲ့ Software Project တွေမှာလည်း အသုံးပြုသင့်ပါတယ်။

နမူနာအနေနဲ့ လက်ရှိလူသုံးအများဆုံး VCS ဖြစ်တဲ့ Git အကြောင်းကိုဖော်ပြခဲ့ပေမယ့် အခြား VCS တွေကို စမ်းသပ် လေ့လာသင့်ပါတယ်။ Subversion (Centralize), Mercurial (Distributed), Bazaar (Distributed) စတဲ့ VCS တွေ ရှိပါသေးတယ်။

Joel Spolsky လို့ခေါ်တဲ့ လူသိများထင်ရှားတဲ့ ပညာရှင်တစ်ဦးဖော်ထုတ်ခဲ့တဲ့ "The Joel Test: 12 Steps to Better Code" ဆိုတဲ့ အချက် (၁၂) ချက်ပါဝင်တဲ့ Software Team တစ်ခုရဲ့ အရည်အသွေးကို တိုင်းတာတဲ့ နည်းလမ်းတစ်ခု ရှိပါတယ်။ အဲ့ဒီ အချက်တွေထဲမှာ နံပါတ် (၁) အချက်က "Do you use source control?" ဆိုတဲ့ အချက်ဖြစ်ပါတယ်။ Source Code တွေစီမံနိုင်တဲ့ စနစ်တစ်ခုကို အသုံးပြုခြင်း၊ မပြုခြင်းဟာ Software Team တစ်ခုရဲ့ Performance ပေါ်မှာ အများကြီး သက်ရောက်မှုရှိစေမှာပဲဖြစ်ပါတယ်။

The Joel Test ထဲက ကျန်အချက်တွေနဲ့အတူ နောက်ထပ်အရေးကြီးတဲ့ အကြောင်းအရာတစ်ခုဖြစ်တဲ့ Issue Tracking System အကြောင်းကို နောက်အခန်းမှာ ဆက်လက်ဖော်ပြပေးသွားပါမယ်။

ပြင်ရတာမခက်ပေမယ့်၊ သိပ်အရေးမကြီးလို့ နောင်မှ
ပြင်မယ်ဆိုပြီးထားလိုက်တဲ့ Bug ကို၊ မေ့ပြီးမပြင်မိလို့
အသုံးပြုသူလက်ထဲရောက်မှ အဲ့ဒီ Bug ပြန်ပေါ်တဲ့အတွက်
ကိုယ့် Software ရဲ့အရည်အသွေးကိုမေးခွန်းထုတ်စရာ
ဖြစ်သွားတာလောက် နောင်တရဖို့ကောင်းတာ မရှိပါဘူး။