**Git**

ເນື່​ອງ​ຈາກ​ທຸກ​ມື້​ນີ້​ແມ່ນໃຊ້ Git ໃນ​ການ​ເຮັດ​ວຽກງານ​ຢູ່​ແລ້ວ ແລະ ​ບາງເທື່ອ​ກໍ​ຕ້ອງ​ສອນ​ເລື່ອງ​ນີ້​ໃຫ້​ກັບ​ສະມາຊິກ​ໃໝ່​ພາຍ​ໃນ​ທີມ ກໍ​ເລຍ​ຕັດ​ສິນ​ໃຈ​ຂຽນ​ບົດ​ຄວາມ​ນີ້​ຂຶ້ນ​ມາ​ ເພື່ອ​ທີ່​ຈະ​ໄດ້​ແຊ​ຄ​ວາມ​ຮູ້​ໃຫ້​ກັບ Developer ທີ່​ສົນ​ໃຈ​ໃນ Git ແຕ່​ຍັງ​ບໍ່​ເຂົ້າໃຈ​ ຫລື​ ບໍ່​ຮູ້​ວ່າ​ຈະ​ເຮັດ​ຄວາມ​ເຂົ້າໃຈ​ແບບໃດ​​ດີ ເພາະ​ທຸກ​ມື້​ນີ້​ໂລກ​ຂອງ Developer ນັ້ນ​ໄປ​ໄວ​ເຫຼືອ​ເກີນ ຫຼາຍ​ໆ​ຢ່າງ​ຖ້າ​ເຮົາ​ບໍ່​ຮູ້​ຈັກ ​ຫລື ​ໃຊ້​ບໍ່​ເປັນ​ກໍ​ອາດຈະ​ເຮັດໃຫ້​ພາດ​ໂອ​ກາດ​ດີ​ໆ​ໃນ​ເສັ້ນ​ທາງ​ນີ້​ກໍ​ເປັນ​ໄດ້

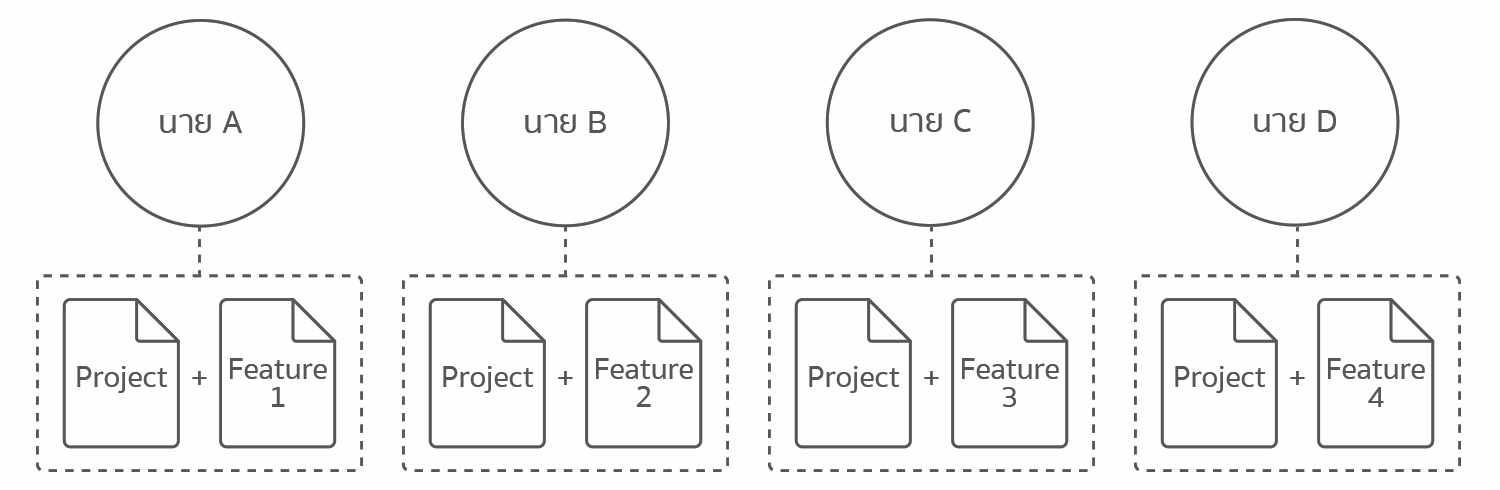
**ເນື້ອຫາ​ຂອງ Git ໃນ​ບົດ​ຄວາມ​ນີ້ ຈະ​ເນັ້ນ​ສຳລັບ​ສາຍ​ໂຄ້​ດ​ເປັນ​ຫຼັກ​**

ມາ​ເວົ້າ​ເຖິງ Version Control Systems (VCS) ກັນ​ກ່ອນ

ໃນ​ຍຸກ​ນີ້​ຕ້ອງ​ບອກ​ເລຍ​ວ່າ Version Control ຖື​ວ່າ​ເປັນ​ໜຶ່ງ​ໃນ​ຄຸນສົມບັດ​ພື້ນ​ຖານ​ຂອງ Developer ກໍ​ວ່າ​ໄດ້ ໂດຍ​ສະເພາະ​ຢ່າງ​ຍິ່ງ ​ບໍລິສັດ​ທີ່​ຕ້ອງ​ເຮັດວຽກ​ຮ່ວມ​ກັນ​ເປັນ​ທີມ (ເຖິງ​ຈະ​ພັດທະນາ​ຄົນ​ດຽວ ກໍ​ຄວນ​ຈະ​ໃຊ້​ຄືກັນ​ນະ)

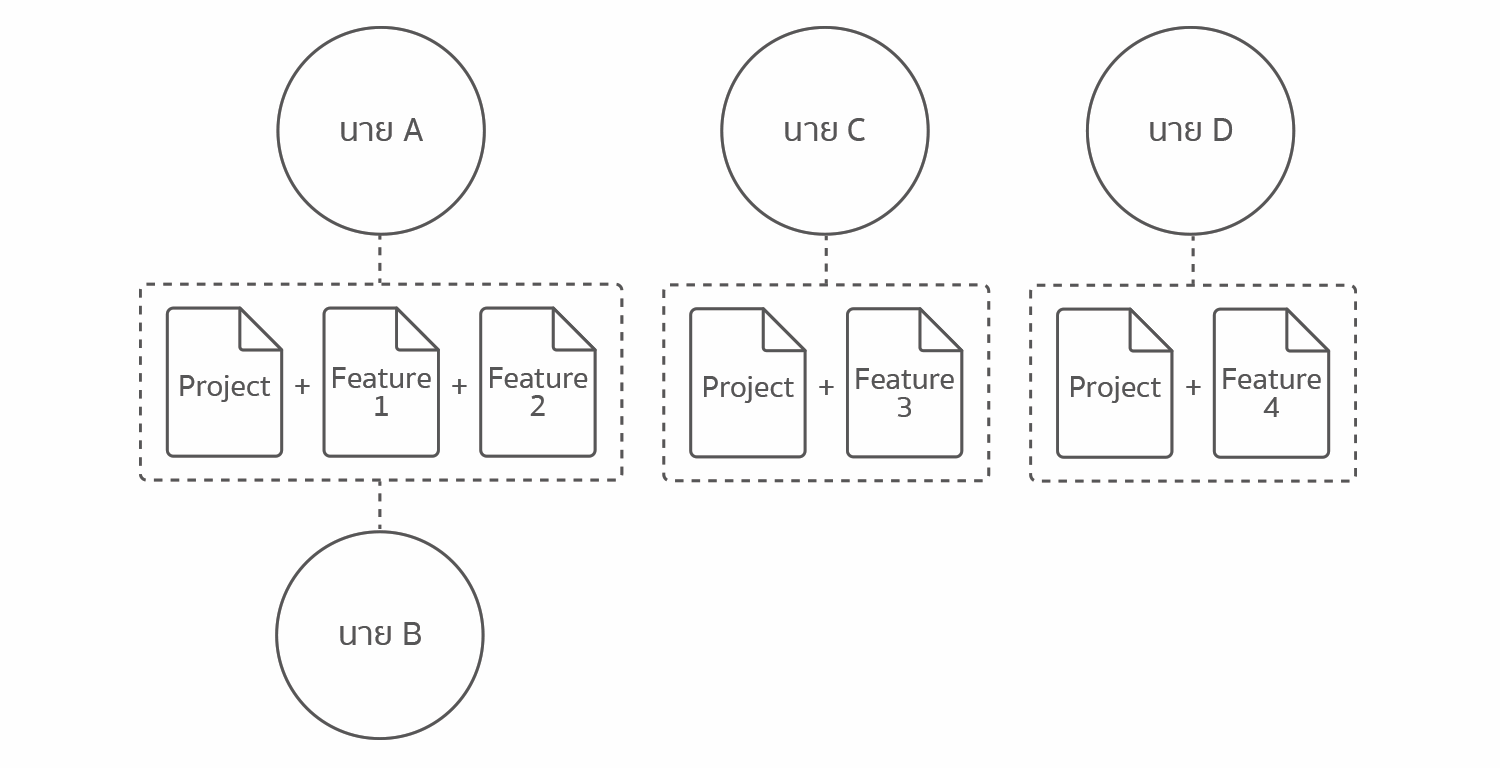
1. **ຄວາມ​ສຳຄັນ​ຂອງ Version Control**

ລອງ​ຄິດ​ເບິ່ງ​ວ່າ​ຖ້າ​ຕ້ອງ​ພັດທະນາ​ໂປຣ​ເຈກ​ຂະໜາດ​ໃຫຍ່​ທີ່​ມີ Developer 4 ຄົົນທີ່​ກຳ​ລັງ​ມະ​ຣຸມ​ມະ​ຕຸ້ມ​ໂຄ້​ດ​ໃນ​ໂປຣ​ເຈກ​ນີ້​ຢູ່ ເຈົ້າ​ຈະ​ໃຊ້​ວິທີ​ໃດ​ເພື່ອ​ເອົາ​ໂຄ້​ດ​ທີ່​ແຕ່​ລະ​ຄົນ​ຂຽນ​ມາ​ທັງຫມົດ​ເຂົ້າ​ນຳກັນ​ໃນ​ໂປຣ​ເຈກ



ວິທີ​​ເກົ່າ​​ທີ່​ໃຊ້​ກັນ​ກໍ​ຄື ກ໋ອບ​ໂປຣ​ເຈກ​ຈາກ​ແຕ່​ລະ​ຄົນ​ມາ​ທັງຫມົດ​ໄວ້​ໃນ​ເຄື່ອງ​ດຽວ​ກັນ ແລ້ວ​ນັ່ງ​ລວມກັນ ມີ​ຄົນ​ໜຶ່ງ​ເຮັດ​ຫນ້າທີ່​ເປີດ​ໂຄ້​ດ​ຂອງ​ແຕ່​ລະ​ຄົນ​ຂຶ້ນ​ມາ ສົມໝຸດວ່າຄົນ​ໆ​ນັ້ນ​ຄື​ ທ້າວ A ແລະ​ ທ້າວ B ເປັນ​ຄົນ​ຂຽນ​ໂຄ້​ດ​ທີ່​ກຳ​ລັງ​ຈະ​ທັງຫມົດ​ໄວ້​ໃນ​ໂປຣ​ເຈກ​ດຽວ​ກັນ ທ້າວ A ກໍ​ເລີຍ​ຕ້ອງ​ຖາມ​ ທ້າວ B ວ່າ​ຂຽນ​ໂຄ້​ດ​ບ່ອນໃດ​ ແລ້ວ​ຄ່ອຍ​ເອົາ​ໄປ​ແປະ​ທັງຫມົດ​ໄວ້​ໃນ​ໂປຣ​ເຈກ​ຫຼັກ

ບັນຫາ​ເກົ່າ​ໆ​ທີ່ເກິດຂື້ນ​ຢູ່​ປະຈຳ​ກໍ​ຄື ໂຄ້​ດ​ທີ່​ ທ້າວ B ໄປ​ແກ້​ໄຂ ພັດ​ໄປ​ທັບ​ຊ້ອນ​ກັບ​ ທ້າວ C ເພາະ​ທ້າວ C ກໍ​ແກ້​ໄຂ​ໂຄ້​ດ​ຈຸດ​ນັ້ນ​ຄືກັນ ກາຍ​ເປັນ​ວ່າ​ ທ້າວ B ກໍ​ຕ້ອງ​ໄປ​ເອີ້ນ​ ທ້າວ C ມາ​ນັ່ງຖາມ​ ເພື່ອ​ບອກ​ໃຫ້​ ທ້າວ A ແກ້​ໄຂ​ໃຫ້​ໂຄ້​ດ​ຂອງ​ ທ້າວ B ແລະ​ ທ້າວ C ເຮັດວຽກ​ຮ່ວມ​ກັນ​ໄດ້



ຍັງ​ບໍ່​ລວມ​ໄປ​ເຖິງ​ກໍລະນີ​ທີ່​ ທ້າວ B ຈື່​ບໍ່​ໄດ້​ວ່າ​ຕັວ​ເອງ​ແກ້​ໄຂ​ໂຄ້​ດ​​ຈຸດ​ໃດ​ແດ່ ເພາະ​ຟີ​ເຈີທີ່​ເຮັດ​ນັ້ນ​ໃຊ້​ເວລາ​ຫຼາຍ​ມື້​ ແລະ ​ຂຽນ​ໂຄ້​ດ​ຫຼາຍ​ບັນ​ທັດ ກາຍ​ເປັນ​ວ່າ​ຕ້ອງ​ນັ່ງ​ລວມກັນ​ເພື່ອ​ໄລ່​ເບິ່ງ​ໂຄ້​ດວ່າ​ພາດ​ຈຸດ​ໃດ​ຫລື​ ບໍ່ ຊຶ່ງ​ລວມປ​ເຖິງ​ບັນຫາ​ອື່ນ​ໆ​ທີ່​ຈະ​ຕາມ​ມາ​ອີກ​ຫລວງ​ຫລາຍ

ສະນັ້ນ​ ຈຶ່ງເຮັດໃຫ້ເກີດຂຶ້ນ​ທີ່​ເອີ້ນວ່າ Version Control ຂຶ້ນ​ມາ​ເພື່ອ​ຄວບ​ຄຸມ​ການ​ປ່ຽນ​ແປງ​ຂອງ​ໂຄ້​ດ​ໃນ​ໂປຣ​ເຈກ ໂດຍ​ປະໂຫຍດ​ທີ່​ເຫັນ​ໄດ້​ແຈ້ງ​ຂອງ Version Control ຈະ​ມີ​ດັ່ງ​ນີ້:

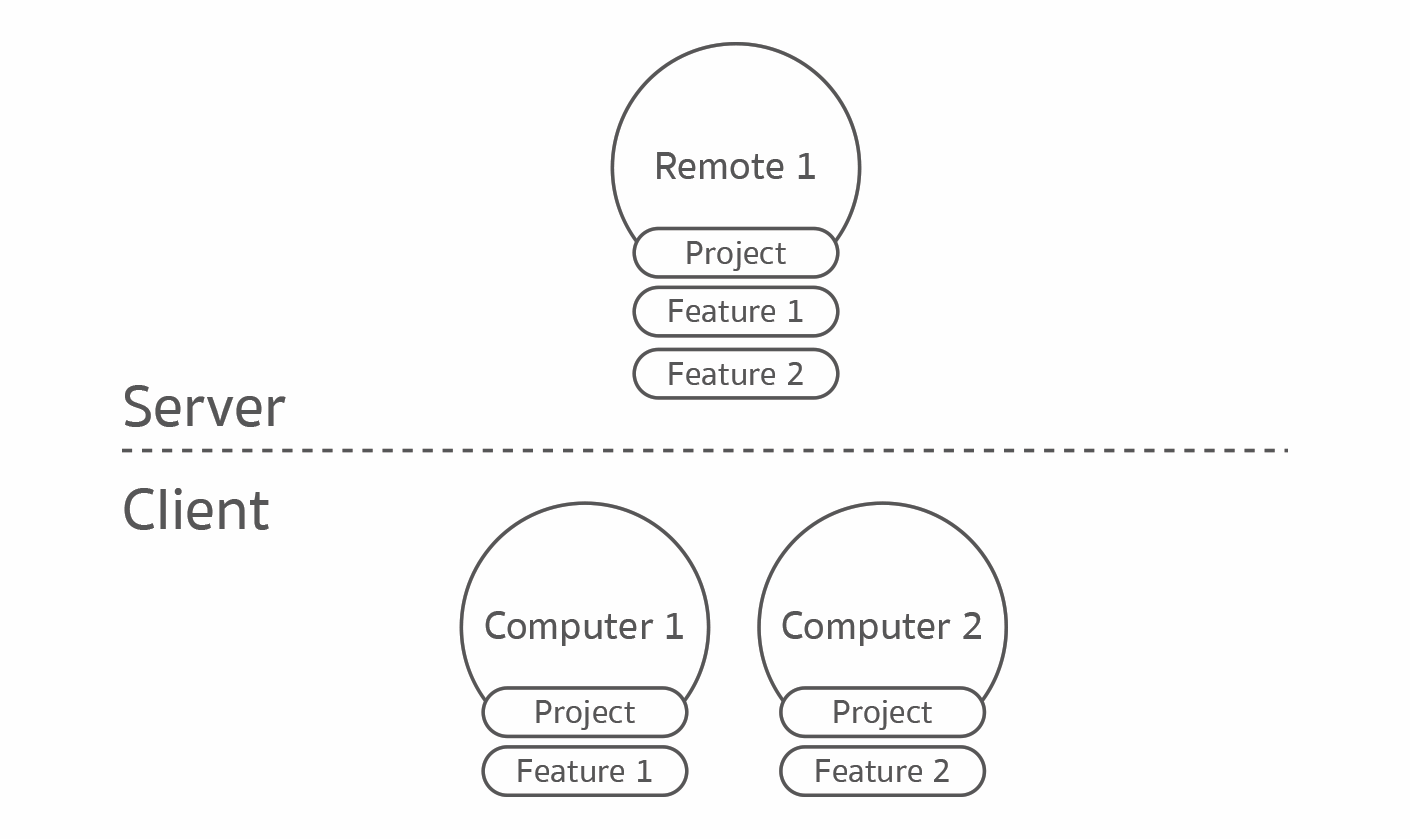
* ເກັບ​ປະຫວັດ​ການ​ແກ້​ໄຂ​ໂຄ້​ດ​ໄວ້​ທຸກ​ເທື່ອ ແລະ​ ຮູ້​ໄດ້​ວ່າ​ໂຄ້​ດ​ຈຸດໃດ​ໃຜ​ເປັນ​ຄົນ​ເພີ່ມ​ເຂົ້າ​ມາ​ ຫລື​ແກ້​ໄຂ
* ຊ່ວຍ​ລວມໂຄ້​ດ​ຈາກ​ຫຼາຍ​ໆ​ຄົນ​ເຂົ້າ​ນຳກັນ​ໃຫ້​ງ່າຍ​ຂຶ້ນ ເບິ່ງ​ໄດ້​ວ່າ​ໂຄ້​ດ​ເດີມ​ຄື​ຫຍັງ ແລະ ​ແກ້​ໄຂ​ເປັນ​ຫຍັງ
* ເມື່ອ​ເກີດ​ບັນຫາ​ກໍ​ສາມາດ​ຕິດ​ຕາມ​ເບິ່ງ​ປະຫວັດ​ການ​ແກ້​ໄຂ​ໂຄ້​ດ​ໃນ​ແຕ່​ລະ​ໄຟ​ລ​ ​ແຕ່​ລະ​ບັນ​ທັດ​ໄດ້​ງ່າຍ
* ຊ່ວຍ​ໃຫ້​ສາມາດ​ຈັດການ​ໂປຣ​ເຈກ​ໄດ້​ຢ່າງ​ເປັນ​ ເບິ່ງ​ໂຄ້​ດ​ແຕ່​ລະ​ສ່ວນ​ເປັນ​ຟີ​ເຈີ ບໍ່​ຂຽນ​ໂຄ້​ດ​ຂ້າມ​ຟີ​ເຈີ​ໄປ​ມາ​ໃນ​ໂຄ້​ດ​ຊຸດ​ດຽວ​ກັນ
* ເປັນ Backup ໄປ​ໃນ​ຕັວ ບໍ່​ຕ້ອງ​ຢ້ານ​ເວລາ​ໂຄ້​ດ​ມີ​ບັນຫາ​ແລ້ວ​ຕ້ອງ Rollback ກັບ​ໄປ​ໃຊ້​ໂຄ້​ດ​ຊຸດ​ເກົ່າ ແລະ ​ໃຊ້​ພື້ນ​ທີ່​ໃນ​ການ​ເກັບ​ຂໍ້​ມູນ​ນ້ອຍ​ເມື່ອ​ທຽບ​ກັບ​ການ Backup ແບບ​ເກັບ​ທັງ​ໂປຣ​ເຈກ​ໄວ້​ທຸກ​ເທື່ອ​ບ່ອນ​ເຮັດວຽກ Backup
* ສາມາດ Track ການ​ເຮັດ​ວຽກງານ​ຂອງ​ທຸກ​ຄົນ​ພາຍ​ໃນ​ທີມ​ໄດ້​ຈາກ History

1. **ການເຮັດວຽກຂອງ Git**

ທີ່ຈິງແລ້ວມັນ​ກໍ​ຄື​ການ​ເຮັດ​ວຽກງານ​ແບບ Distributed Version Control .ໃຫ້​ລອງ​ຄິດເຖິງ​ພາ​ບວ່າ​ມີ Server ກາງ​ເຄື່ຶອງ​ໜຶ່ງ​ທີ່​ລໍ​ຖ້າ​ເກັບ​ຂໍ້​ມູນ​ຈາກ​ຜູ້​ໃຊ້​ແຕ່​ລະ​ເຄື່ອງ​ກ່ອນ​

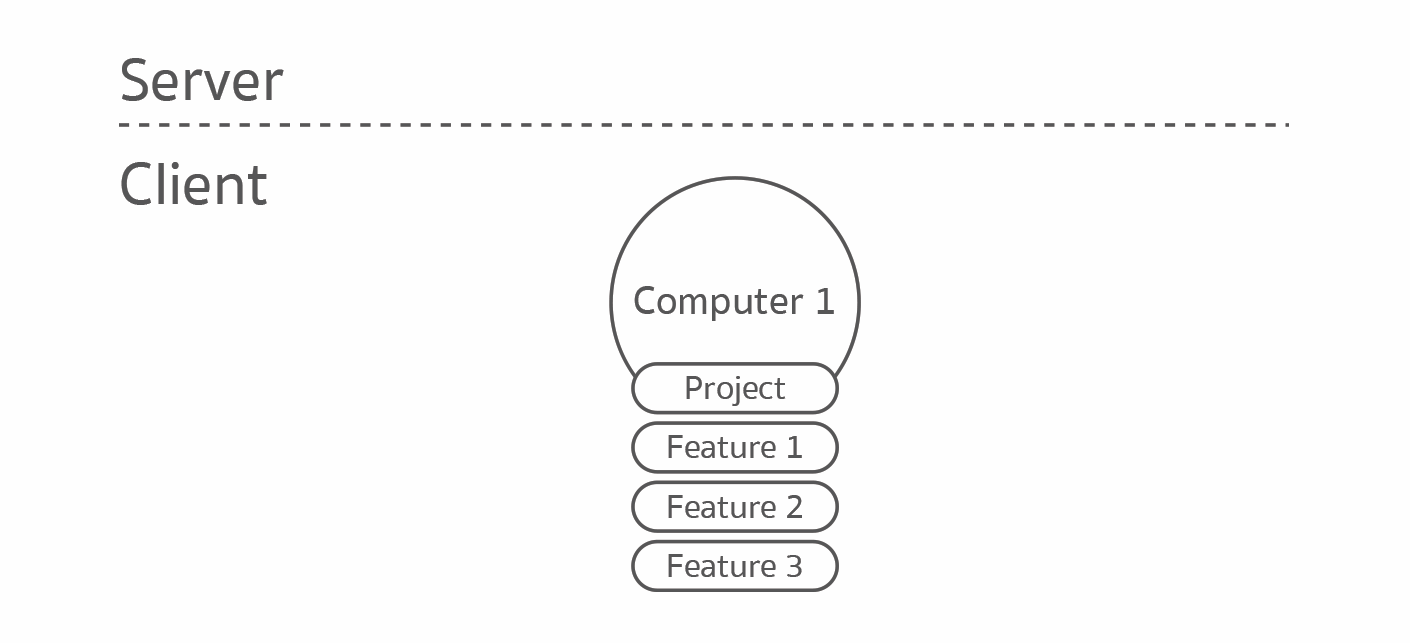


Server ກາງ​ເອີ້ນວ່າ Remote 1 ສ່ວນ Developer ທີ່​ເຮັດ​ໂປຣ​ເຈກ​ນີ້​ຢູ່​ຈະ​ຮຽກວ່າ Computer 1 ແລະ 2 ຊຶ່ງ​ແຕ່​ລະ​ຄົນ​ກໍ​ເຮັດ​ຄົນ​ລະ Feature ຢູ່ ຊຶ່ງ Feature ຂອງ​ແຕ່​ລະ​ຄົນ​ນັ້ນ​ກໍ​ຈະ​ຖືກ​ເກັບ​ໄວ້​ທີ່ Remote 1

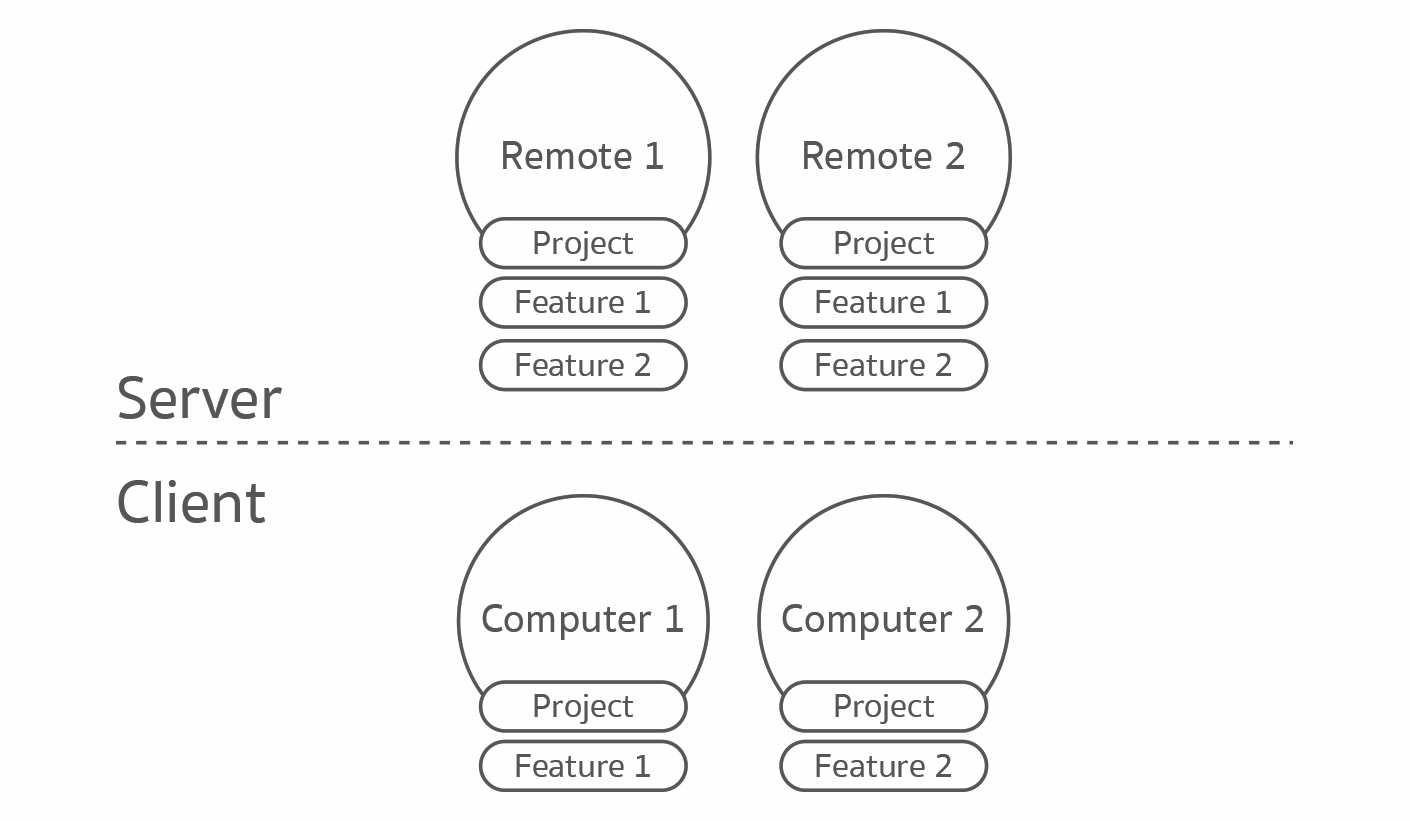


ອາດຈະ​ເບິ່ງ​ຄື​ວ່າ​ມັນ​ເປັນ​ລະບົບ Server ກາງ​ທີ່​ຄອຍ​ຖ້າ Backup ຂໍ້​ມູນ​ຂອງ​ທຸກ​ຄົນ​ຢູ່​ຕະຫລອດ​ເວລາແຕ່​ໃນ​ຄວາມ​ເປັນ​ຈິງ Git ນັ້ນ​ມີ​ຫຍັງ​ຫລາຍກວ່າ​ນີ້​ອີກ​ຫຼາຍຢ່າງ

Git ອອກ​ແບບ​ມາ​ໃຫ້​ເຮັດວຽກ​ກະ​ຈາຍ​ແບບ​ບໍ່​ມີ​ສູນ​ກາງ ທຸກ​ເຄື່ອງ​ເຮັດວຽກ​ເປັນ VCS ດ້ວຍ​ຕັວ​ເອງ​ໄດ້. ນັ້ນ​ໝາຍ​ຄວາມ​ວ່າ​ບໍ່​ຈຳ​ເປັນ​ຕ້ອງ​ມີ Server ກາງ​ກໍ​ໄດ້ ສາມາດ​ໃຊ້​ເຄື່ອງ​ສ່ວນ​ຕົວ​ເຮັດ​ເປັນ VCS ໄດ້​ເລີຍ ແຕ່​ຖ້າ​ຕ້ອງ​ເຮັດວຽກ​ຮ່ວມ​ກັນ​ຫຼາຍ​ໆ​ເຄື່ອງ ກໍ​ຕ້ອງ​ໃຊ້ Server ເປັນ​ຕົວ​ກາງ​ໃນ​ການ​ທັງຫມົດ​ຂໍ້​ມູນ



ຊຶ່ງ​ຈະ​ເຮັດໃຫ້​ຂໍ້​ມູນ​ຂອງ​ເຮົາ​ບໍ່​ຜູກ​ຂາດ​ກັບ Server ຈົນ​ເກີນ​ໄປ ໃນ​ເວລາ​ທີ່ Server ກາງ​ມີ​ບັນຫາ​ຫລື​ ວ່າ​ເຮັດວຽກ​ແບບ Offline ເຮົາ​ກໍ​ຍັງ​ຄົງ​ເຮັດວຽກ​ໄດ້​ຢູ່ ໂດຍ​ໃຊ້​ຂໍ້​ມູນ​ຈາກ VCS ພາຍ​ໃນ​ເຄື່ອງ​ຕົວ​ເອງ ພໍ​ເຊື່ອມ​ຕໍ່​ກັບ Server ກາງ​ກໍ​ຄ່ອຍ Sync ຂໍ້​ມູນ​ພາຍ​ຫລັງ​ໄດ້ ແລະ​ ເມື່ອ​ທຸກ​ໆ​ເຄື່ອງ​ເຮັດວຽກ​ເປັນ VCS ຢູ່​ແລ້ວ ຈຶ່ງ​ເຮັດໃຫ້​ເຮົາ​ສາມາດ​ມີ Server ກາງ​ຫລາຍກວ່າ 1 ເຄຶ່ອງ​ໄດ້​ເຊັ່ນ​ກັນ

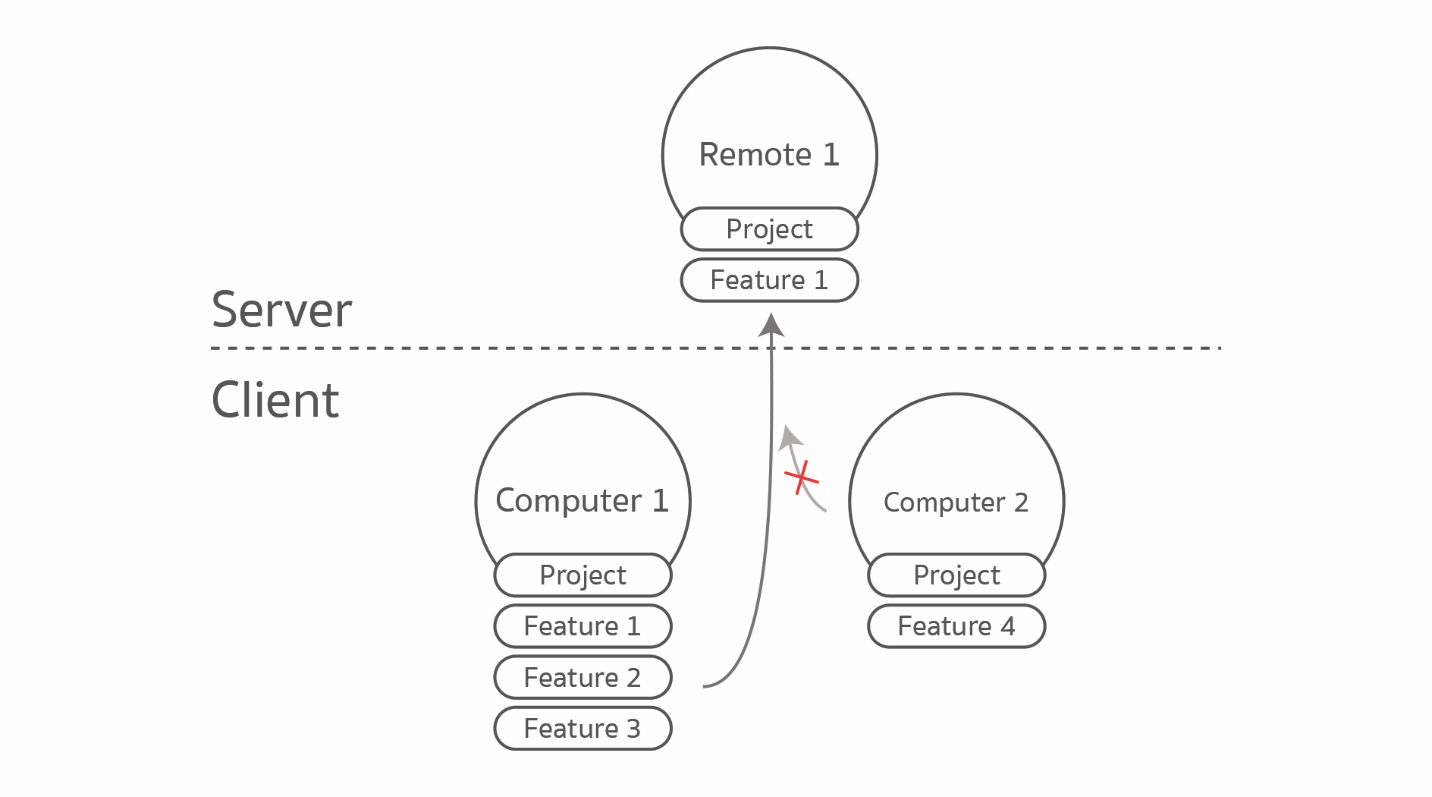


1. **Git ຈະ Sync ຂໍ້​ມູນ​ເມື່ອ​ເຮົາ​ສັ່ງ​ເທົ່າ​ນັ້ນ**

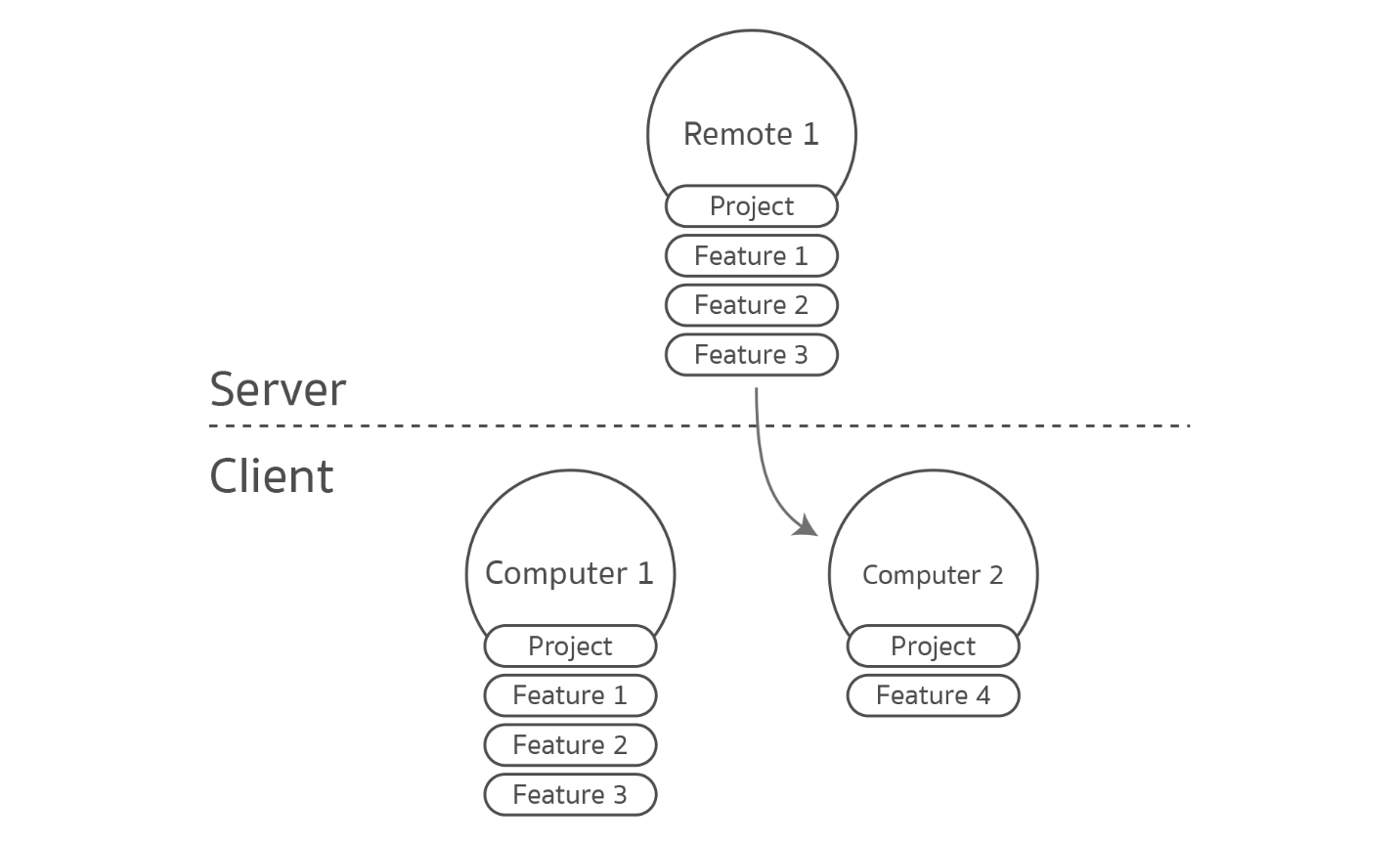
ຂັ້ນ​ຕອນ​ການ Sync ຂໍ້​ມູນ​ຂອງ Git ​ຈະ​ເວົ້າ​​ໃນ​ພາຍ​ຫຼັງ ຂໍ​ເນັ້ນ​ໄປ​ທີ່​ພາ​ບລວມ​ຂອງ​ການ​ເຮັດ​ວຽກ​ກ່ອນ

ເວລາ​ທີ່​ເຮົາ​ຢາກ​ຈະ Sync ຂໍ້​ມູນ​ຈາກ Remote ມາ​ລົງ​ເຄື່ອງ ຫລື​ ຈາກ​ເຄື່ອງ​ສົ່ງ​​ໄປ​ Remote ກໍ​ຈະ​ຕ້ອງ​ເປັນ​ຄົນ​ຈັດການ​ເອງ ຈະບໍ່​ໄດ້​ Sync ໃຫ້​ໂດຍ​ອັດ​ຕະໂນ​ມັດ ເພາະ​ວ່າ​ມັນ​ຈະ​ມີ​ເລື່ອງ​ຈຸກ​ຈິກ​ທີ່​ເຮົາ​ຕ້ອງ​ເປັນ​ຄົນ​ຕັດ​ສິນ​ໃຈ ​ແລະ ​ເລືອກ​ເອງ ບໍ່​ສາມາດ​ໃຫ້​ຕັວ Git ຈັດການ​ໃຫ້​ອັດ​ຕະໂນມັດ​ໄດ້

ຍົກ​ຕົວ​ຢ່າງ​ເຊັ່ນ ໂຄ້​ດ​ຈາກ​ນັກ​ພັດທະນາ 2 ຄົົນທີ່​ດັນ​ໄປ​ແກ້​ໄຂ​ທີ່​ບັນ​ທັດ​ດຽວ​ກັນ ແຕ່​ຄຳ​ສັ່ງ​ຕ່າງ​ກັນ ແລະ​ ຈຸດປະສົງ​ຂອງ​ໂຄ້​ດ​ຕ່າງ​ກັນ ແລ້ວ Git ຈະ​ໄປ​ຕັດສິນ​ໄດ້​ແບບໃດ​​ວ່າ​ຕ້ອງ​ເອົາ​ໂຄ້​ດ​ແບບໃດ



ແລະ​ ນັ້ນ​ກໍ​ຄື​ຫນ້າທີ່​ຂອງ​ນັກ​ພັດທະນາ​ທີ່​ຈະ​ຕ້ອງ​ເປັນ​ຄົນ​ຕັດ​ສິນ​ໃຈ ​ຫລື ​ແກ້​ໄຂ​ໂຄ້​ດ​ນັ້ນ​ເອງ​ ໂດຍ​ຄົົນທີ່ Sync ຂໍ້​ມູນ​ຂຶ້ນ Remote ກ່ອນ​ຈະ​ບໍ່​ຕ້ອງ​ເຮັດ​ຫຍັງ ແລະ ​ຄົົນທີ່ Sync ຂໍ້​ມູນ​ພາຍ​ຫລັງ​ຈະ​ຕ້ອງ​ເປັນ​ຄົນ​ຈັດການ​ເອງ ຊຶ່ງ​ສາມາດ​ຕັດ​ສິນ​ໃຈ​ໄດ້​ວ່າ​ຈະ​ໃຊ້​ໂຄ້​ດ​ຂອງ​ຄົນທຳອິດ ຫລື​ ຈະ​ໃຊ້​ໂຄ້​ດ​ຂອງ​ຕົວ​ເອງ ຫລື​ວ່າ​ຈະ​ແກ້​ໄຂ​ໂຄ້​ດ​ເພື່ອ​ໃຫ້​ຮອງ​ຮັບ​ໂຄ້​ດ​ຂອງ​ທັງ​ສອງ​ຄົນ​ກໍ​ເຮັດ​ໄດ້​ເຊັ່ນ​ກັນ

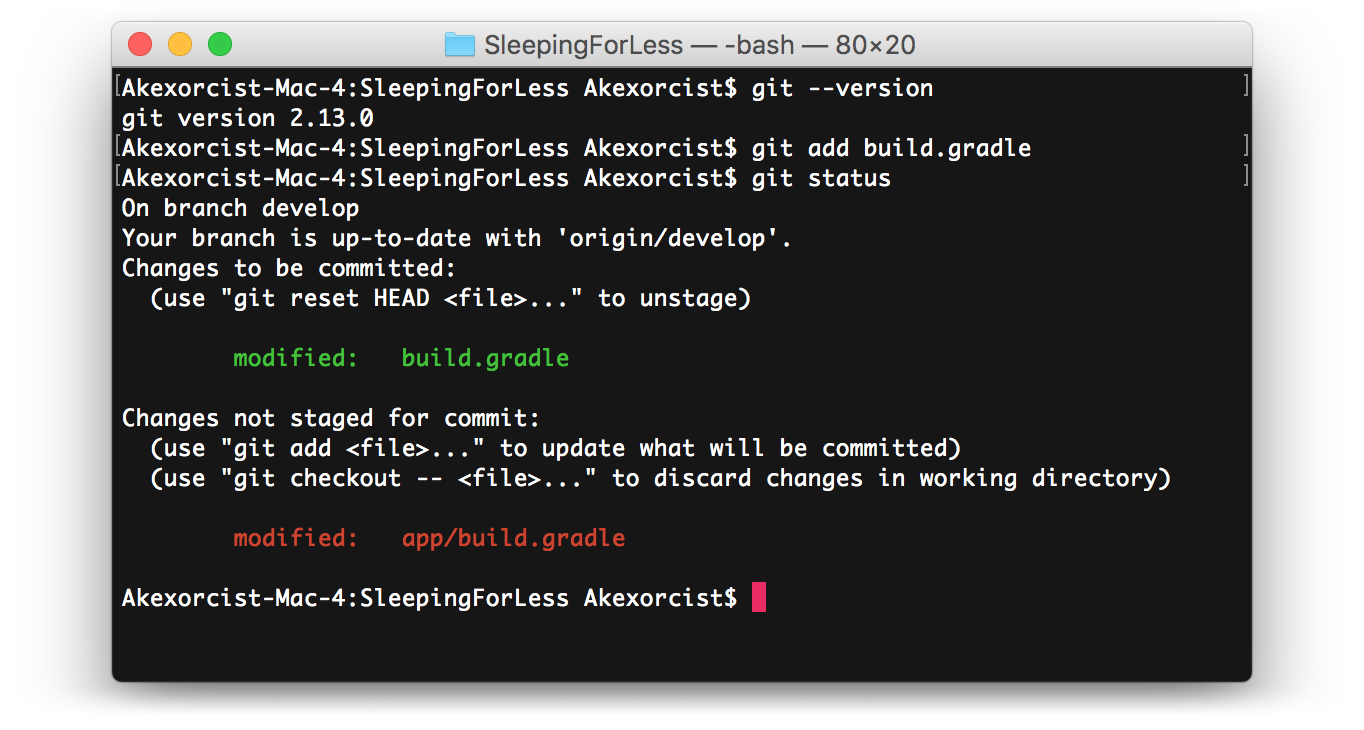


ດັ່ງ​ນັ້ນ Git ຈະ​ບໍ່​ໄດ້​ເຮັດວຽກ​ອັດ​ຕະໂນມັດ​ໂດຍ​ສົມບູນ​ແບບ ເພາະ​ມັນ​ຍັງມີ​ເລື່ອງ​ລະອຽດ​ລະ​ອ່ອນ​ອີກ​ຫຼາຍ​ໆ​ຢ່າງ​ທີ່​ຜູ້​ໃຊ້​ຕ້ອງ​ເຂົ້າໃຈ​ເພື່ອ​ທີ່​ຈະ​ໄດ້​ຈັດການ​ກັບ​ໂຄ້​ດ​ໃນ​ໂປຣ​ເຈກ​ໄດ້​ຢ່າງ​ເໝາະ​ສົມ

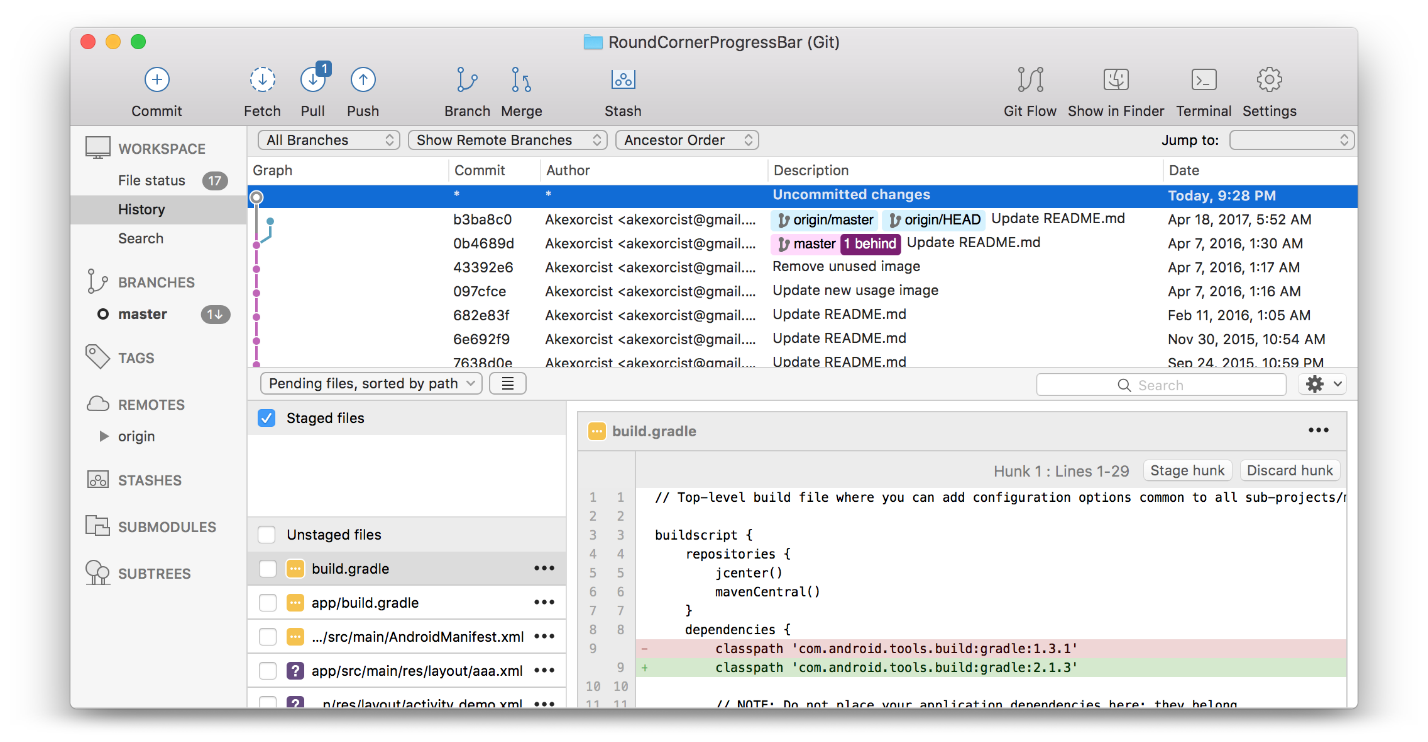
1. **ການ​ເອີ້ນ​ໃຊ້​ງານ Git**

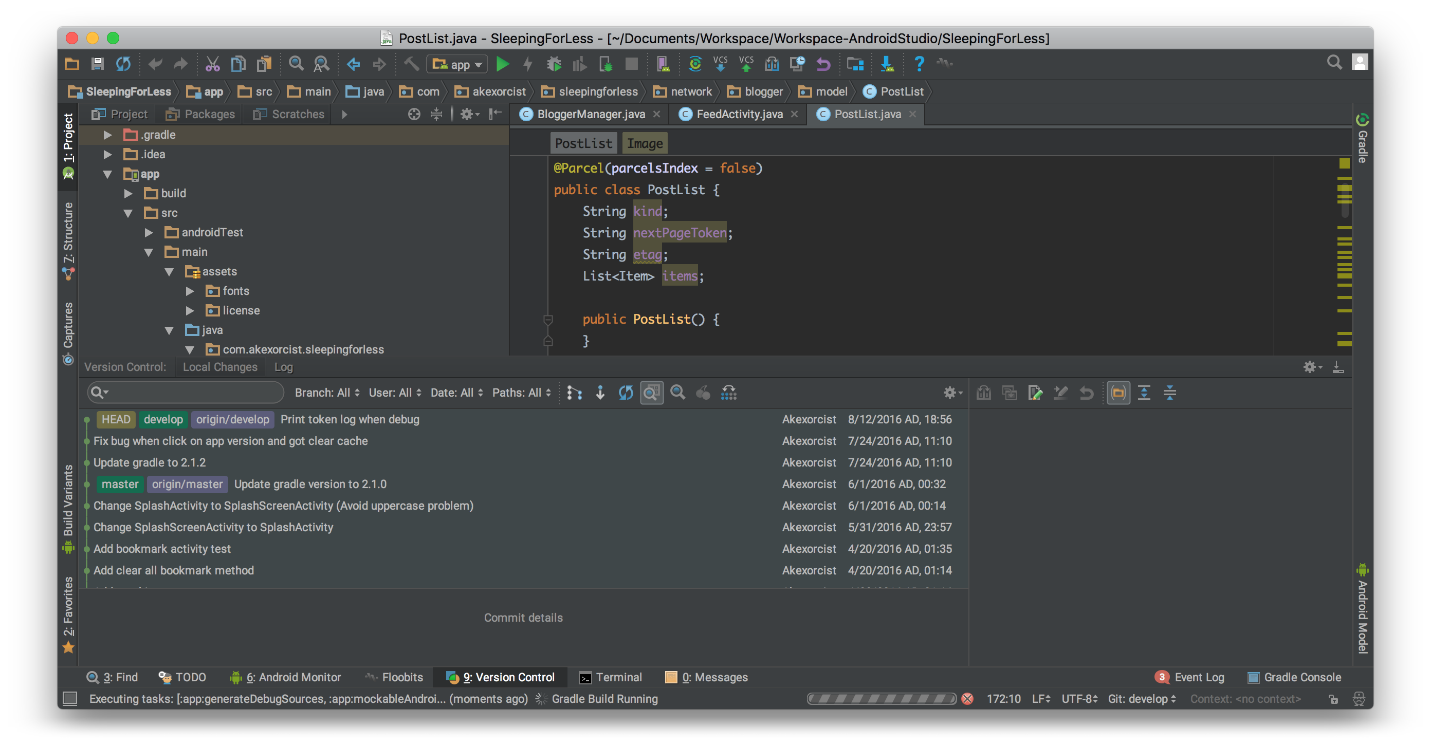
ມີ​ຢູ່ 2 ແບບ​ຫຼັກ​ໆ​ກໍ​ຄື Command Line ກັບ GUI

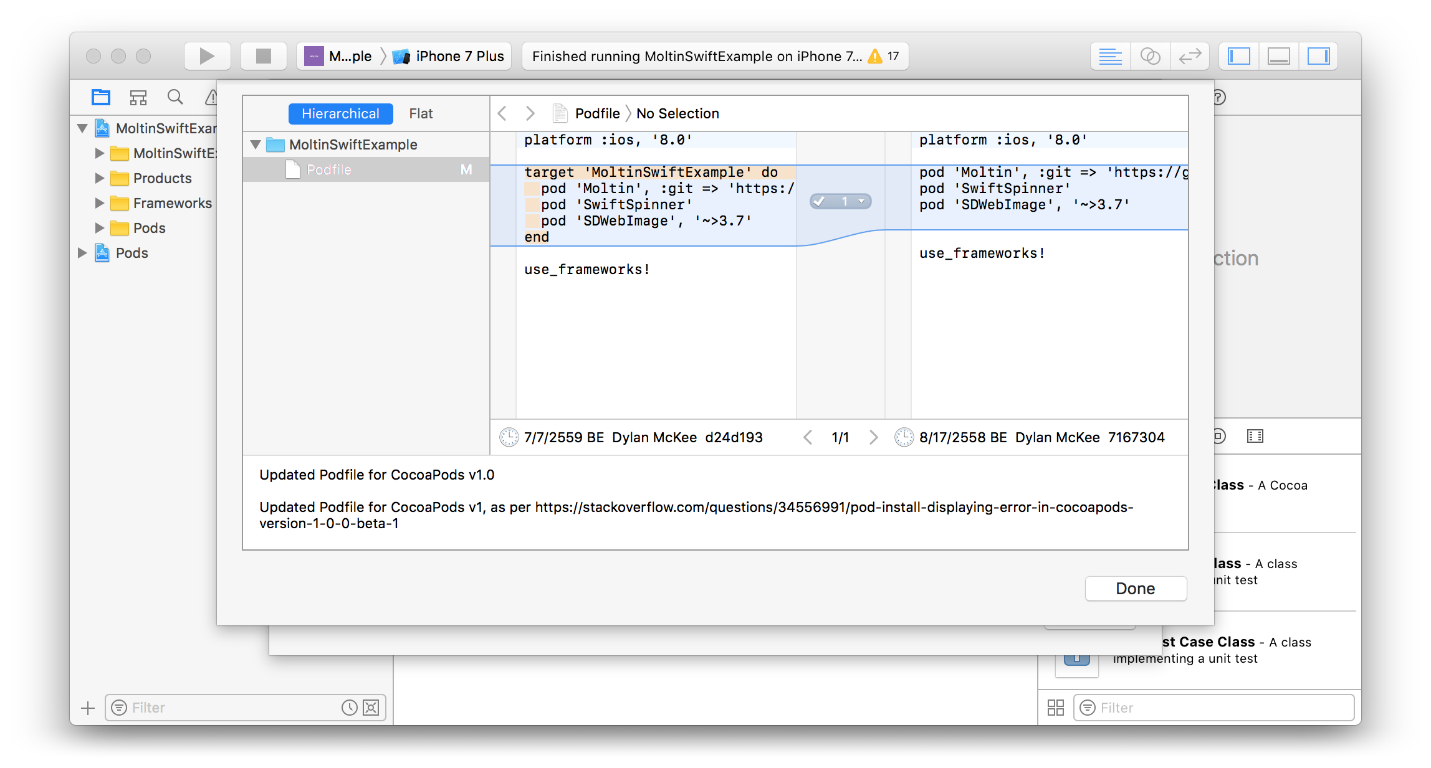
* Command Line : ພິມ​ຄຳ​ສັ່ງ​ຂອງ Git ຈາກ Terminal ຫລື Command Prompt ໂດຍ​ກົງ​ນັ້ນ​ເອງ



* GUI : ໃຊ້​ໂປຣ​ແກຣມ​ພວກ Git GUI ຢ່າງ [SourceTree](https://www.sourcetreeapp.com/), [TortoiseGit](https://tortoisegit.org/" \t "_blank) ຫຼື [GitHub Desktop](https://desktop.github.com/) เປັນ​ຕົ້ນ



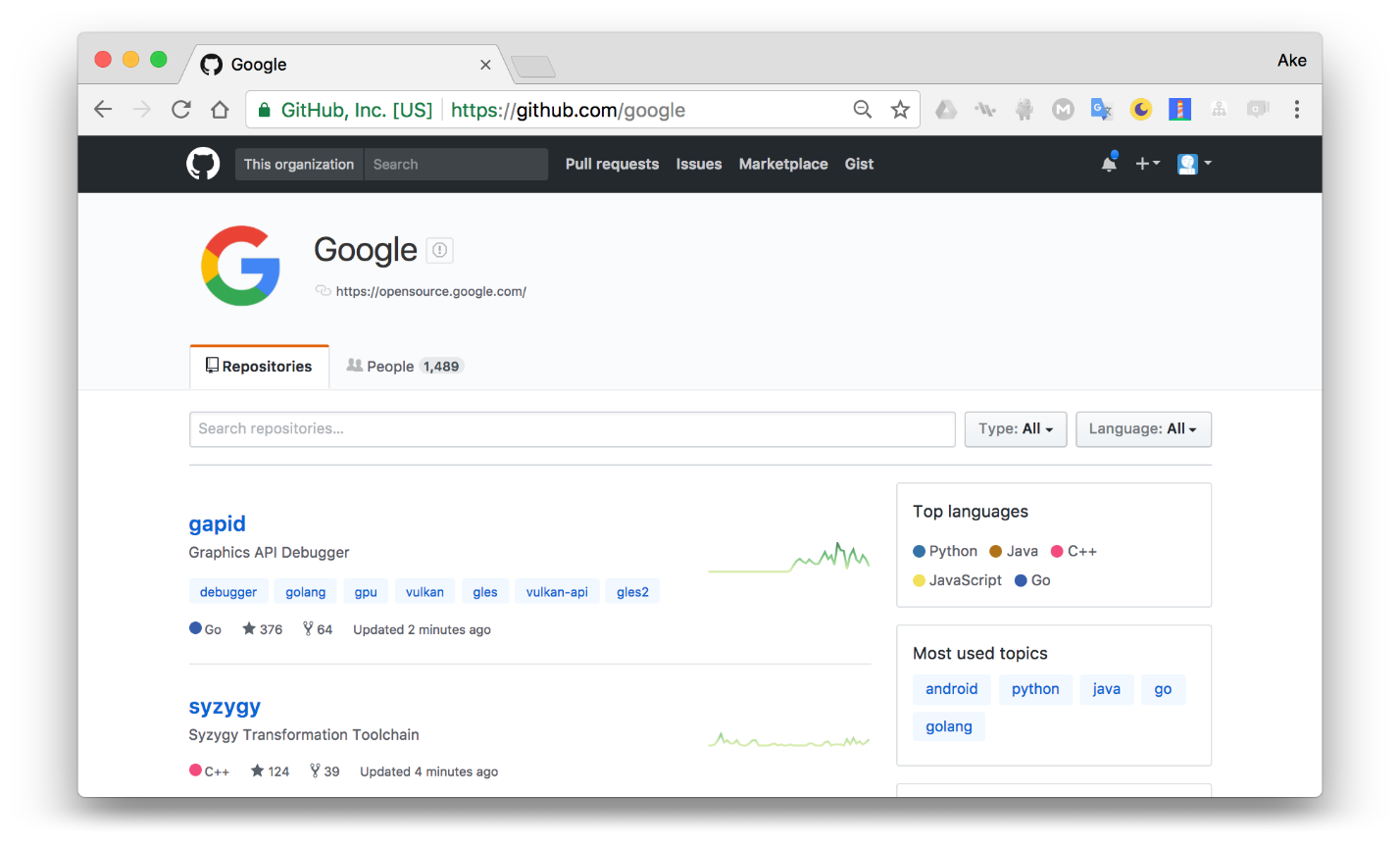
* ແລະ IDE ບາງ​ຕັວ​ກໍ​ມີ Git GUI ໃຫ້​ໃນ​ຕົວ​ເລີຍ​ ບໍ່​ຕ້ອງ​ໄປ​ລົງ​ເພີ່ມ​ໃຫ້​ເສຍ​ເວລາ



# Server ສຳລັບ​ໃຫ້​ບໍລິການ Git (Version Control Repository Hosting Service)

ການ​ໃຊ້ Git ໃຫ້​ເກີດ​ປະໂຫຍດ​ທີ່ສຸດ​​ຕ້ອງ​ມີ Server ທີ່​ເຮັດ​ຫນ້າທີ່​ເປັນ VCS ຊຶ່ງ​ໃນ​ປະຈຸບັນ​ນີ້​ກໍ​ມີ​ທາງ​ເລືອກ​ຫລວງ​ຫລາຍ​ບໍ່​ວ່າ​ຈະ​ໃຊ້​ບໍລິການ​ຈາກ​ເວັບ​ຕ່າງ​ໆ​ ຫລື ​ຕັ້ງ Server ຂຶ້ນ​ມາ​ເອງ

ໃນ​ກໍລະນີ​ເລີ່ມ​ຕົ້ນ​ ​ຂໍ​ແນະ​ນຳ​ໃຫ້​ໃຊ້ Server ທີ່​ໃຫ້​ບໍລິການ Git ໄປ​ເລີຍ ເພາະ​ເຮົາ​ບໍ່​ຕ້ອງ​ຈັດການ​ຫຍັງ​ຫລາຍ ພຽງແຕ່​ສະໝັກ​ກໍ​ເຂົ້າໄປ​ໃຊ້​ງານ​ໄດ້​ເລີຍ ໂດຍ​ເວັບ​ໄຊ​ທີ່​ນິຍົມ​ກໍ​ຈະ​ມີ​ດັ່ງ​ນີ້: [GitHub](https://github.com/), [Bitbucket](https://bitbucket.org/product" \t "_blank), [Gitlab](https://about.gitlab.com/" \t "_blank). ຊຶ່ງ​ເວັບ​ເຫຼົ່າ​ນີ້​ຈະ​ມີ​ໃຫ້​ບໍລິການ​ທັງ​ແບບ Public (ໃຜ​ໆ​ກໍ​ເຂົ້າ​ມາ​ເບິ່ງ​ໂຄ້​ດ​ຂອງ​ເຮົາ​ໄດ້)ແລະ Private (ສະເພາະ​ຄົົນທີ່​ເຮົາ​ອະນຸຍາດ​ເທົ່າ​ນັ້ນ) ຖ້າ​ຢາກ​ເຮັດ​ເປັນ Open Source ຢູ່​ແລ້ວ ກໍ​ແນະ​ນຳ GitHub



# ຄຳ​ຕ່າງ​ໆ​ທີ່​ຈະ​ຕ້ອງ​ຮູ້​ຈັກ​ເມື່ອ​ໃຊ້​ງານ Git (ຊຸດ​ທີ່ 1)

ຂໍ​ແບ່ງ​ເປັນ​ຫຼາຍ​ໆ​ຊຸດ ເພື່ອ​ທີ່​ຈະ​ໄດ້​ອະທິບາຍ​ການ​ເຮັດ​ວຽກງານ​ເທື່ອ​ລະ​ສ່ວນ​ໃຫ້​ເຂົ້າໃຈ​ໄດ້​ງ່າຍ​ຂຶ້ນ

* Repository
* Clone
* Commit
* Unstaged
* Staged
* Push
* Pull
* Fetch
* Conflict
* Merge Commit

1. **Repository**

ເວລາ​ທີ່​ເຮົາ​ພັດທະນາ​ໂປຣ​ແກຣມ​ ເຮົາ​ຈະ​ຕ້ອງ​ສ້າງ​ສິ່ງ​ທີ່ເອີ້ນວ່າ Project ຊຶ່ງ​ການສ້າງ Project ສຳລັບ​ໃຊ້​ງານ Git ຈະ​ເອີ້ນວ່າ Repository

ຊຶ່ງ Repository ຂອງ Git ແທ້ໆ​ແລ້ວ​ກໍ​ຄື Folder ທີ່​ໃຊ້​ເກັບ​ຂໍ້​ມູນ​ນັ້ນ​ເອງ ຢາກ​ຈະ​ເກັບ​ຫຍັງ​ໄວ້​ໃນ​ນັ້ນ​ກໍ​ຍັດ​ເຂົ້າໄປ​ໄດ້​ເລີຍ (ໃນ​ຄວາມ​ເປັນ​ຈິງ 1 Repository ສາມາດ​ເກັບ Project ເທົ່າໃດ​ກໍ​ໄດ້​ຕາມ​ທີ່​ຕ້ອງ​ການແຕ່​ສ່ວນ​ໃຫ່ຍ​ກໍ​ນິຍົມ​ເກັບ Project 1 ຕົວ​ຕໍ່ 1 Repository)

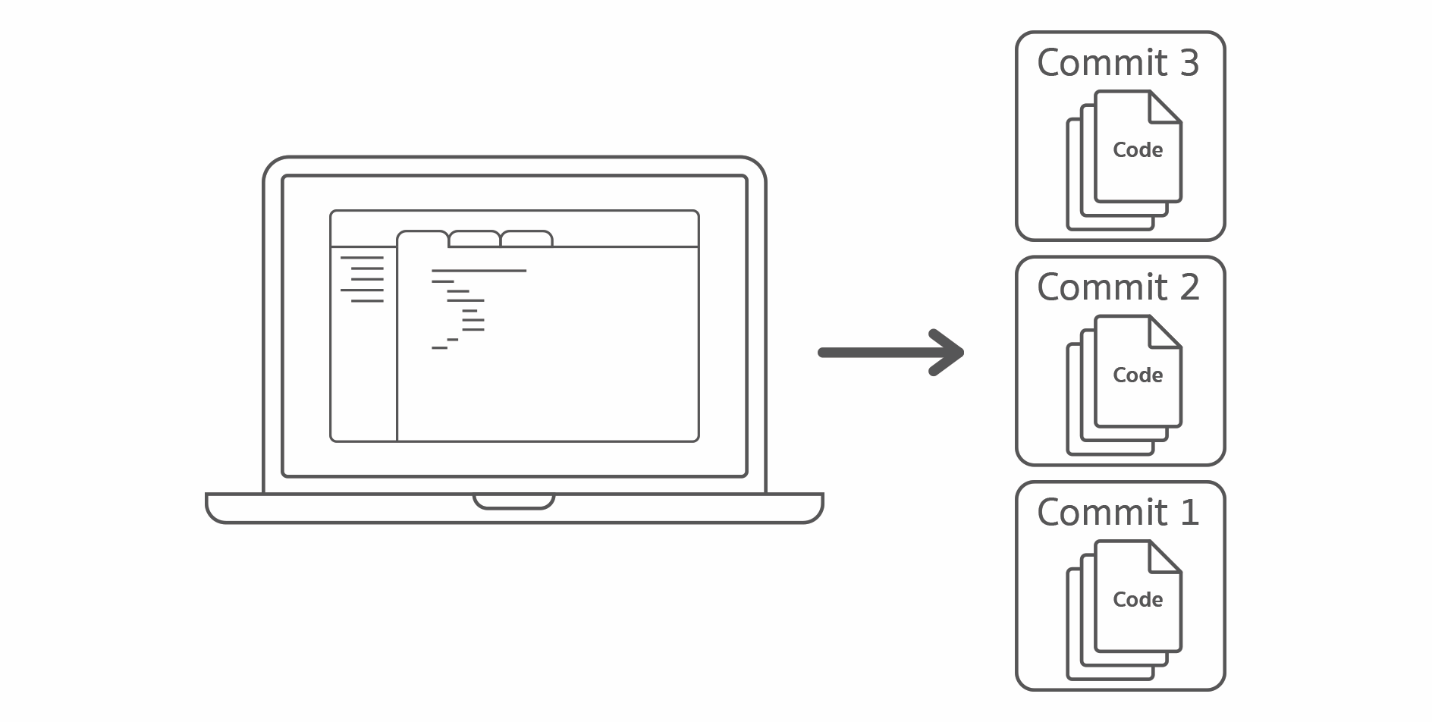
ເວລາ​ທີ່​ຜູ້​ອ່ານ​ເປີດ​ເບິ່ງ​ໂຄ້​ດ​ຕ່າງ​ໆ​ຂອງ​ນັກ​ພັດທະນາ​ເທິງ GitHub (ຍົກ​ຕົວ​ຢ່າງ​ເຊັ່ນ <https://github.com/google/volley>) ໃນ​ແຕ່​ລະ​ໂປຣ​ເຈກ​ທີ່​ເຮົາ​ເຫັນ​ ທີ່​ເອີ້ນວ່າ Repository

## **Clone**

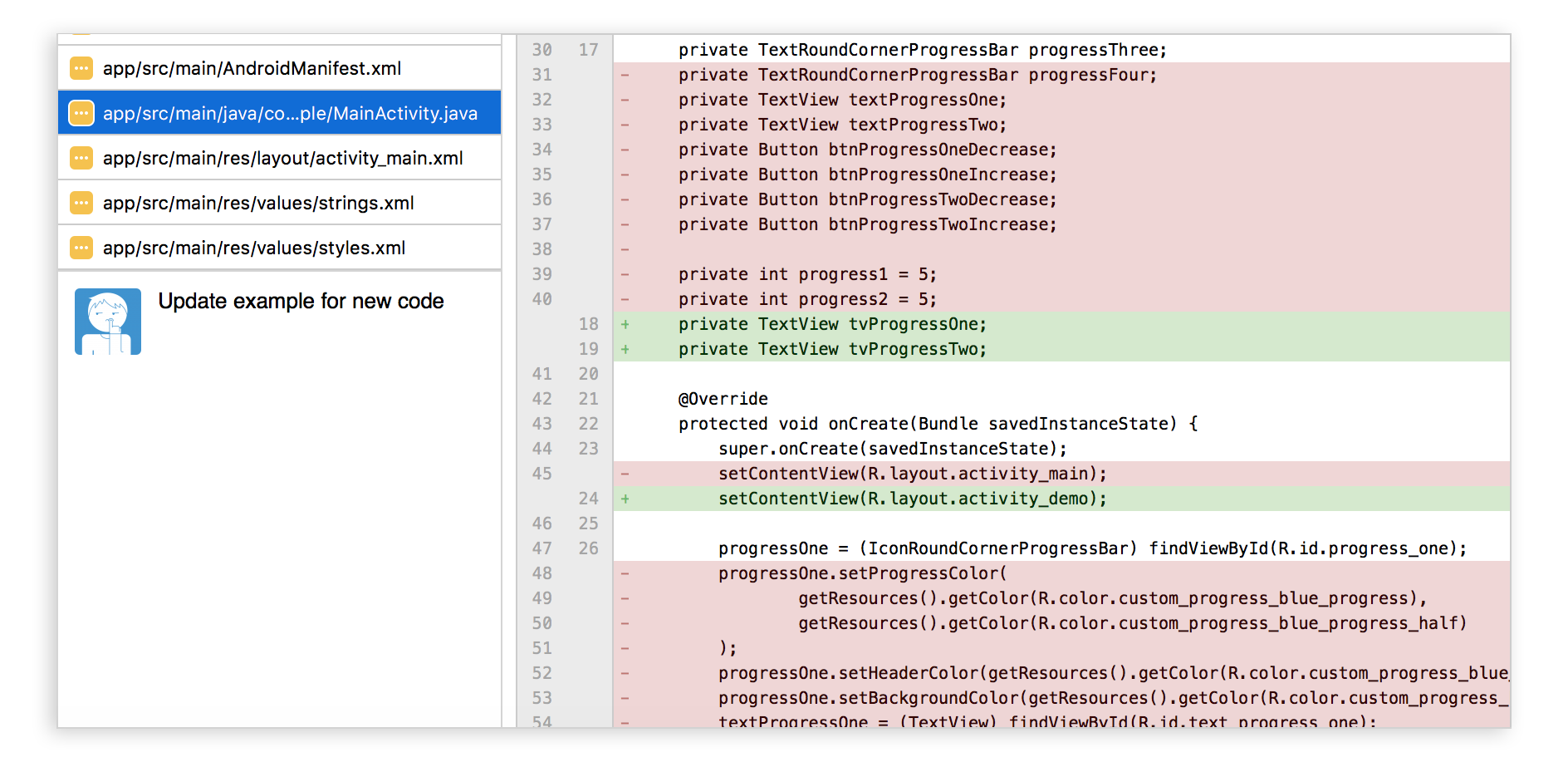
ເວລາ​ທີ່​ຜູ້​ອ່ານ​ມີ Repository ຢູ່​ເທິງ Remote ​ຢູ່​ແລ້ວ ແລະ​ ຕ້ອງ​ການ Sync ມາ​ລົງ​ເຄື່ອງ​ຂອງ​ເຮົາ ເຮົາ​ຈະ​ຕ້ອງ​ເຮັດ​ສິ່ງ​ທີ່​ເອີ້ນວ່າ CloneRepository ຫລື​ ກໍ​ຄື​ການ​ກ໋ອບ Repository ຈາກ Remote ມາ​ລົງ​ເຄື່ອງ​ນັ້ນ​ເອງ

## **Commit**

ເວລາ​ທີ່​ມີ​ຂໍ້​ມູນ​ທີ່​ແກ້​ໄຂ​ສຳເລັດ​ແລ້ວ (ໂຄ້​ດ​ທີ່​ຂຽນ​ຄຳ​ສັ່ງ​ບາງຢ່າງ​ສຳເລັດ​ແລ້ວ) ແລ້ວ​ຢາກ​ຈະ​ກໍ່ການ Backup ເກັບ​ໄວ້​ໃນ VCS ຈະ​ຮຽກ​ກັນ​ວ່າ Commitຊຶ່ງ​ການ Commit ຈະ​ສາມາດ​ເລືອກ​ໄດ້​ວ່າ​ຈະ​ເອົາ​ໄຟ​ລ​​ໃດ​ອີກ (ບໍ່​ຈຳ​ເປັນ​ຕ້ອງ​ເລືອກ​ທຸກ​ໄຟ​ລ​)



ຊຶ່ງ​ເບື້ອງ​ຫຼັງ​ຂອງ Commit ກໍ​ຄື​ໃນ​ແຕ່​ລະ​ເທື່ອ​ທີ່​ເຮັດວຽກ Commit ມັນ​ຈະຈື່ແຕ່ວ່າ​ມີ​ຈຸດໃດ​ຂອງ​ຂໍ້​ມູນ​ທີ່​ຖືກ​ປ່ຽນ​ແປງ​ໄປ​ເລື້ອຍ​ໆ ດັ່ງ​ນັ້ນ​ໃນ​ແຕ່​ລະ Commit ມັນ​ຈະ​ບໍ່​ມີ​ໄຟ​ລ​​ຂໍ້​ມູນ​ສະບັບ​ເຕັມ ແຕ່​ມັນ​ສາມາດ​ຍ້ອນ​ເບິ່ງ History ໄດ້​ວ່າ​ມີ​ການ​ແກ້​ໄຂ​ຫຍັງ​ແດ່ ເຮັດໃຫ້​ຮູ້​ວ່າ​ໃນ Commit ນັ້ນ​ໆ ແຕ່​ລະ​ໄຟ​ລ​​ມີ​ຂໍ້​ມູນ​ເປັນ​ແບບໃດ



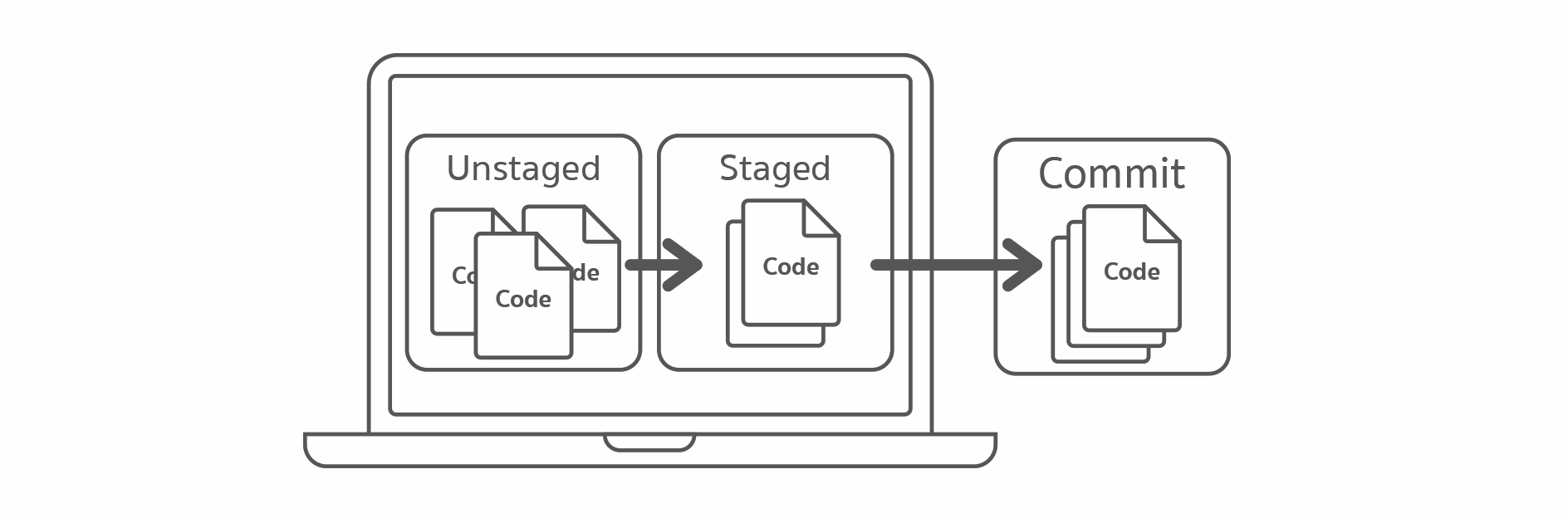
ແລະ ​ໃນ​ການ Commit ແຕ່​ລະ​ເທື່ອ ຈະ​ຕ້ອງ​ໃສ່ Commit Message ເພື່ອ​ອະທິບາຍ​ລາຍ​ລະອຽດ​ຂອງ​ຂໍ້​ມູນ​ໃນ Commit ນັ້ນ​ໆ​ວ່າ​ເຮົາ​ເຮັດ​ຫຍັງ​ໄປ​ແດ່ ເພື່ອ​​ມາ​ເບິ່ງ​ໃນ​ພາຍ​ຫຼັງ​ຈະ​ໄດ້​ອ່ານ​ຈາກ Commit Message ໄດ້​ເລິຍ ບໍ່​ຕ້ອງ​ໄປ​ນັ່ງ​ກົດ​ເບິ່ງ​ເອງ​ເທື່ອ​ລະ​ໄຟ​ລ​ ດັ່ງ​ນັ້ນ​ການ Commit ທີ່​ດີ​ຈຶ່ງ​ຄວນ​ໃສ່​ໃຈ​ກັບ Commit Message ນຳອີກ

## **Unstaged ແລະ Staged**

ຂໍ​ເວົ້າ​ເຖິງ 2 ຄຳ​ນີ້​ໄປ​ພ້ອມ​ໆ​ກັນ​ເລີຍ

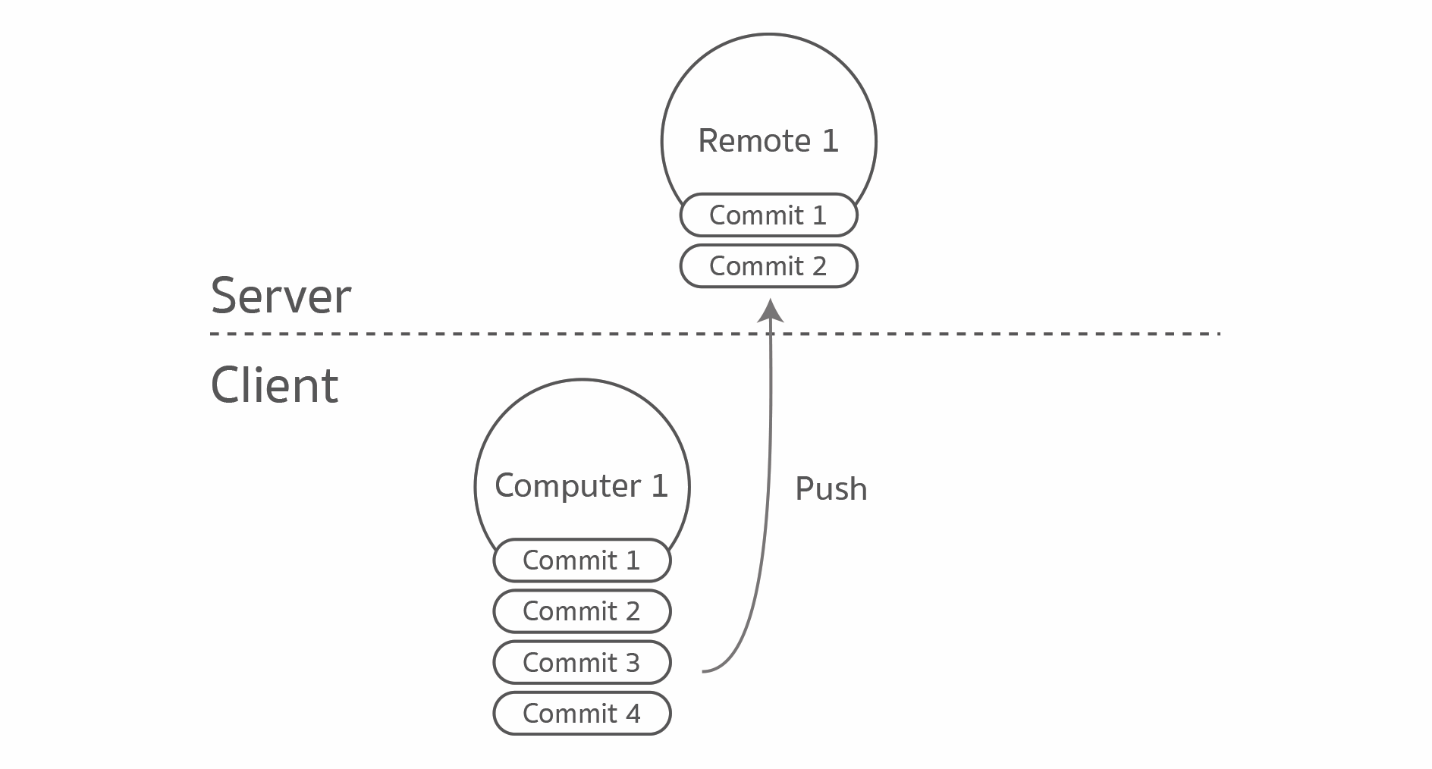
ເວລາ​ເຮົາ​ແກ້​ໄຂ​ໂຄ້​ດ​ ຫລື ​ແກ້​ໄຂ​ຂໍ້​ມູນ ໄຟ​ລ​ທີ່​ຖືກ​ແກ້​ໄຂ​ຈະ​ຢູ່ໃນ​ສ​ຖາ​ນະ Unstagedແລະ​ ເວລາ​ທີ່​ເຮົາ​ເຮັດ​ຫຍັງ​ສຳເລັດ​ຮຽບຮ້ອຍ ແລ້ວ​ຢາກ​ຈະ Commit ເກັບ​ໄວ້ ຈະ​ຕ້ອງ​ເລືອກ​ໄຟ​ລ​​ທີ່​ຕ້ອງ​ການ​ເພື່ອ​ຍ້າຍ​ເຂົ້າ​ສູ່​ໃນ​ສ​ຖາ​ນະ Stagedກ່ອນ​ເຖິງ​ຈະ​ກໍ່ການ Commit ໄດ້

ຊຶ່ງ​ສະ​ຖາ​ນະ Unstaged ແລະ Staged ເຮັດໃຫ້​ເຮົາ​ສາມາດ​ເລືອກ​ສະເພາະ​ບາງ​ໄຟ​ລ​​ສຳລັບ Commit ໄດ້​ນັ້ນ​ເອງ ຈະ​ໄດ້ Commit ສະເພາະ​ໄຟ​ລ​​ທີ່​ເຮົາ​ຂຽນ​ສຳເລັດ​ແລ້ວ



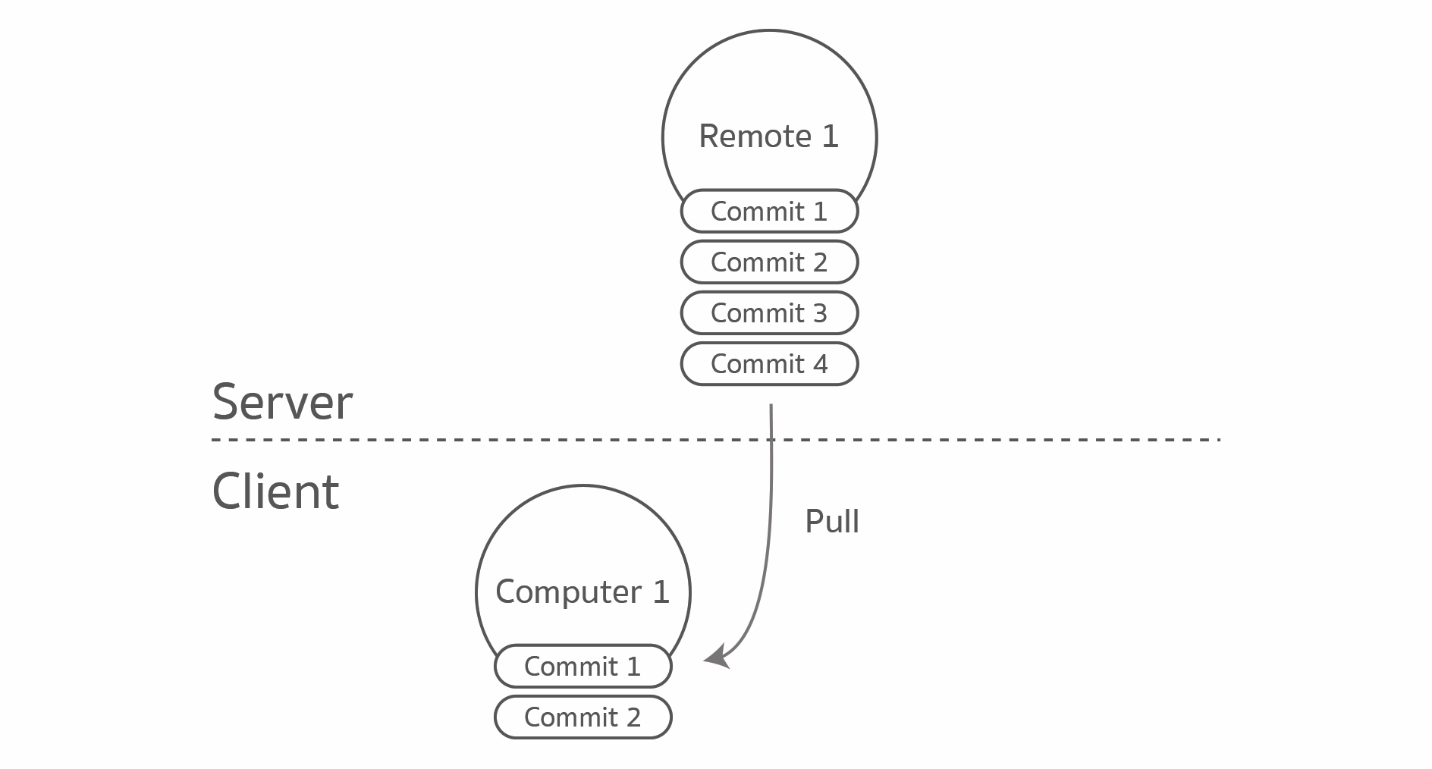
1. **Push**

ເວລາ​ທີ່​ມີ Commit ຢູ່ໃນ​ເຄື່ອງ​ ແລ ະ​ຕ້ອງ​ການ​ຈະ Sync ຂຶ້ນ​ໄປ​ເກັບ​ໄວ້​ໃນ Remote ຈະ​เอี้นຂັ້ນ​ຕອນ​ນີ້​ວ່າ Push



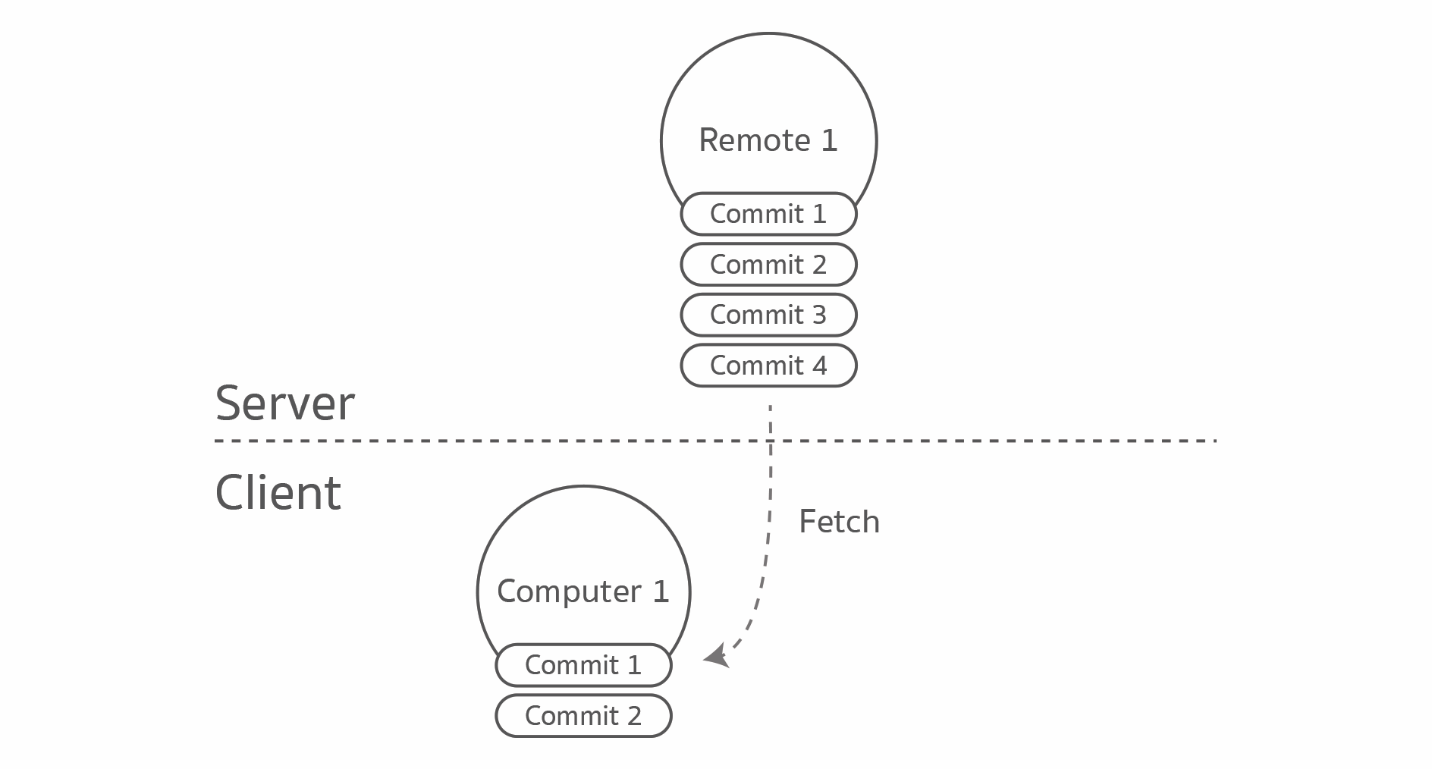
1. **Pull**

ເວລາ Sync ຈາກ Remote ເພື່ອ​ດຶງ​ຂໍ້​ມູນ Commit ໃໝ່​ໆ​ລົງ​ມາ​ເກັບ​ໄວ້​ໃນ​ເຄື່ອງ​ຈະ​ຮຽກ​ຂັ້ນ​ຕອນ​ນີ້​ວ່າ Pull



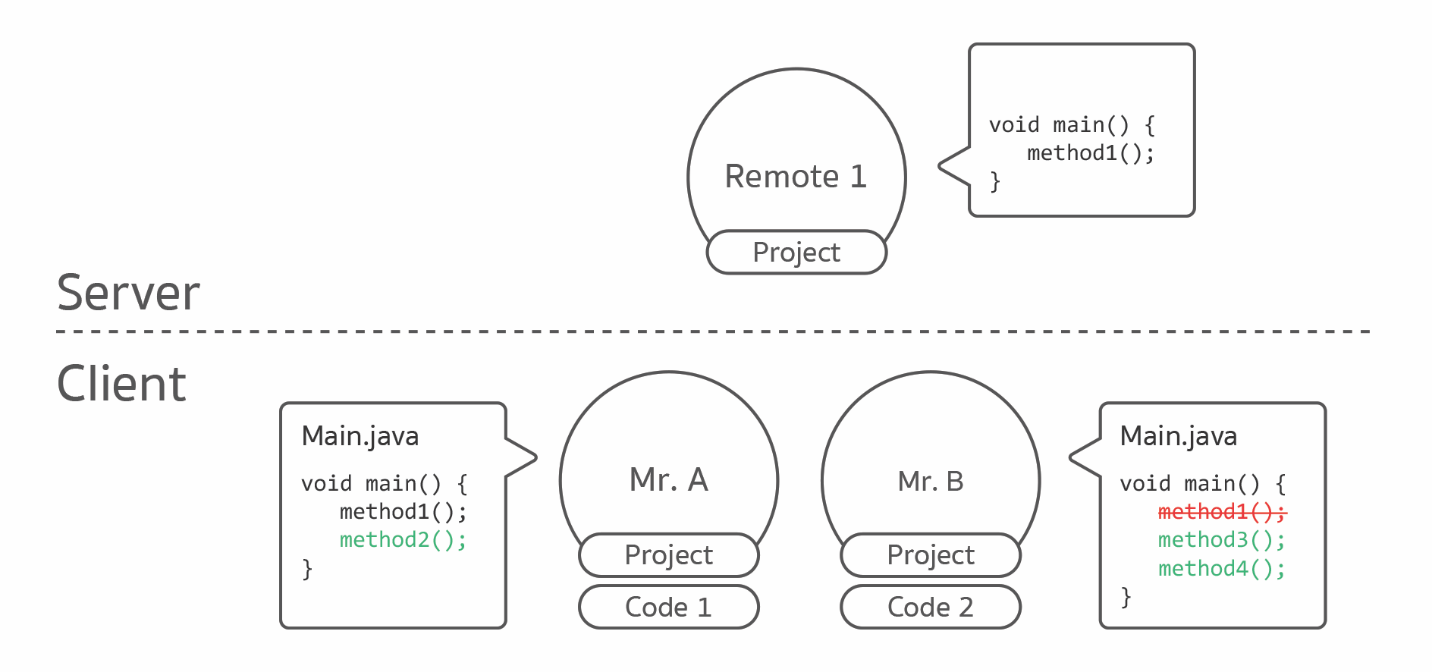
## **Fetch**

ໃນ​ບາງເທື່ອ​ເຮົາ​ອາດຈະ​ບໍ່​ຕ້ອງ​ການ Pull ຂໍ້​ມູນ​ລົງ​ມາ​ເກັບ​ໄວ້​ໃນ​ເຄື່ອງ​ທັນ​ທີ ແຕ່​ຢາກ​ເຊັກ​ສະ​ຖາ​ນະ​ຂອງ Remote ຊື່ໆ​ວ່າ​ມີ​ໃຜ Push ຂໍ້​ມູນ​ໃໝ່​ຂຶ້ນ​ໄປ​ທີ່ Remote ຫລື ​ບໍ່ ເຮົາ​ເອີ້ນວິທີ​ນີ້​ວ່າ Fetch ຊຶ່ງ​ການ Fetch ຈະ​ເຮັດໃຫ້​ເຮົາ​ສາມາດ​ອັບ​ເດດ​ ແລະ​ ເບິ່ງ History ທັງ​ໝົດ​ທີ່​ຢູ່​ເທິງ Remote ໄດ້ໂດຍ​ບໍ່​ຕ້ອງ​ດຶງ​ຂໍ້​ມູນ​ລົງ​ມາ. ຊຶ່ງ​ທີ່ຈິງ​ແລ້ວ​ຂັ້ນ​ຕອນ​ການ Fetch ຈະ​ຖືກ​ເອີ້ນ​ທຸກ​ທີ່​ເຮັດວຽກ Pull. ດັ່ງ​ນັ້ນ ເຮົາ​ຈຶ່ງ​ສາມາດ​ເລືອກ​ໄດ້​​ວ່າ​ຈະ Pull ເລິຍ (​ມັນ​ກໍຈະ Fetch ເພື່ອ​ເຊັກ​ເອງ​) ຫລື​ ຈະ Fetch ເບິ່ງ​ກ່ອນ​ວ່າ​ມີ Commit ຫຍັງ​ເພີ່ມ​ເຂົ້າ​ມາແລ້ວ​ຄ່ອຍ Pull ລົງ​ມາ

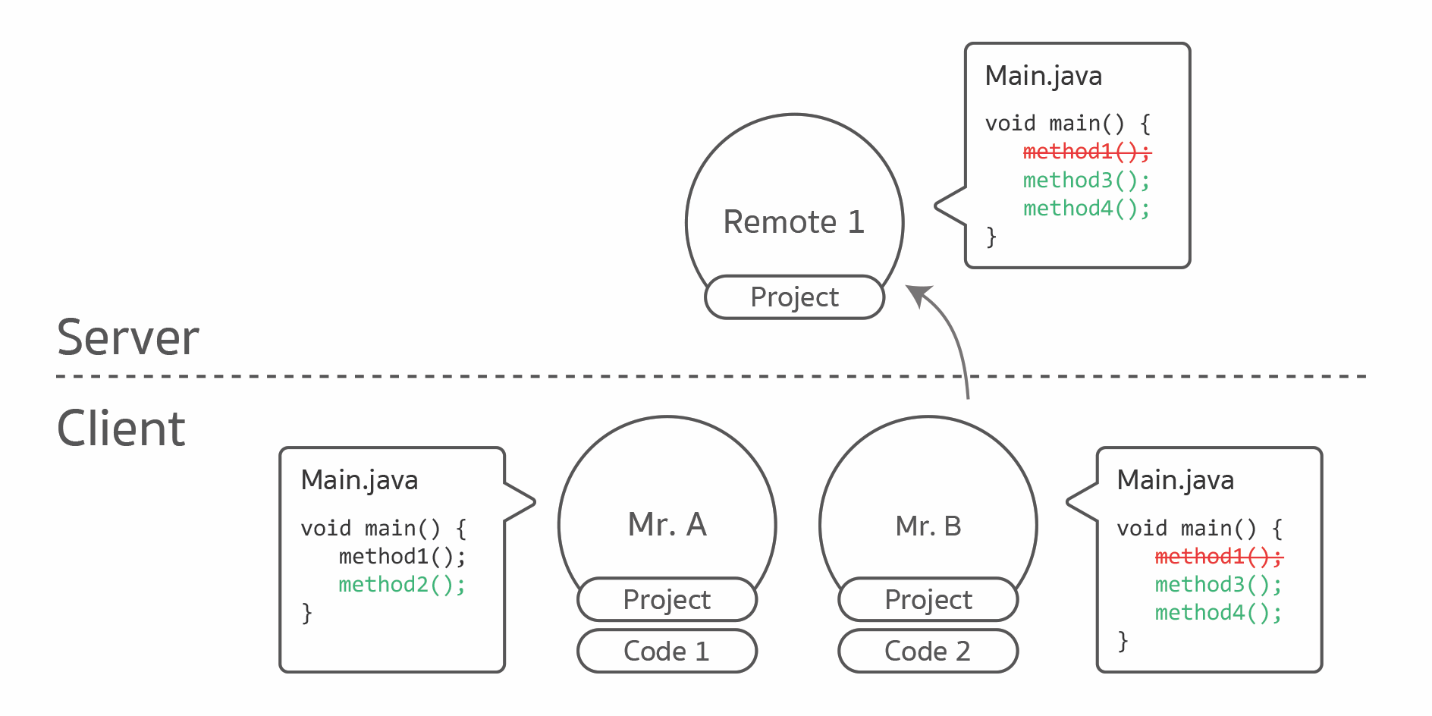


## **Merge Commit**

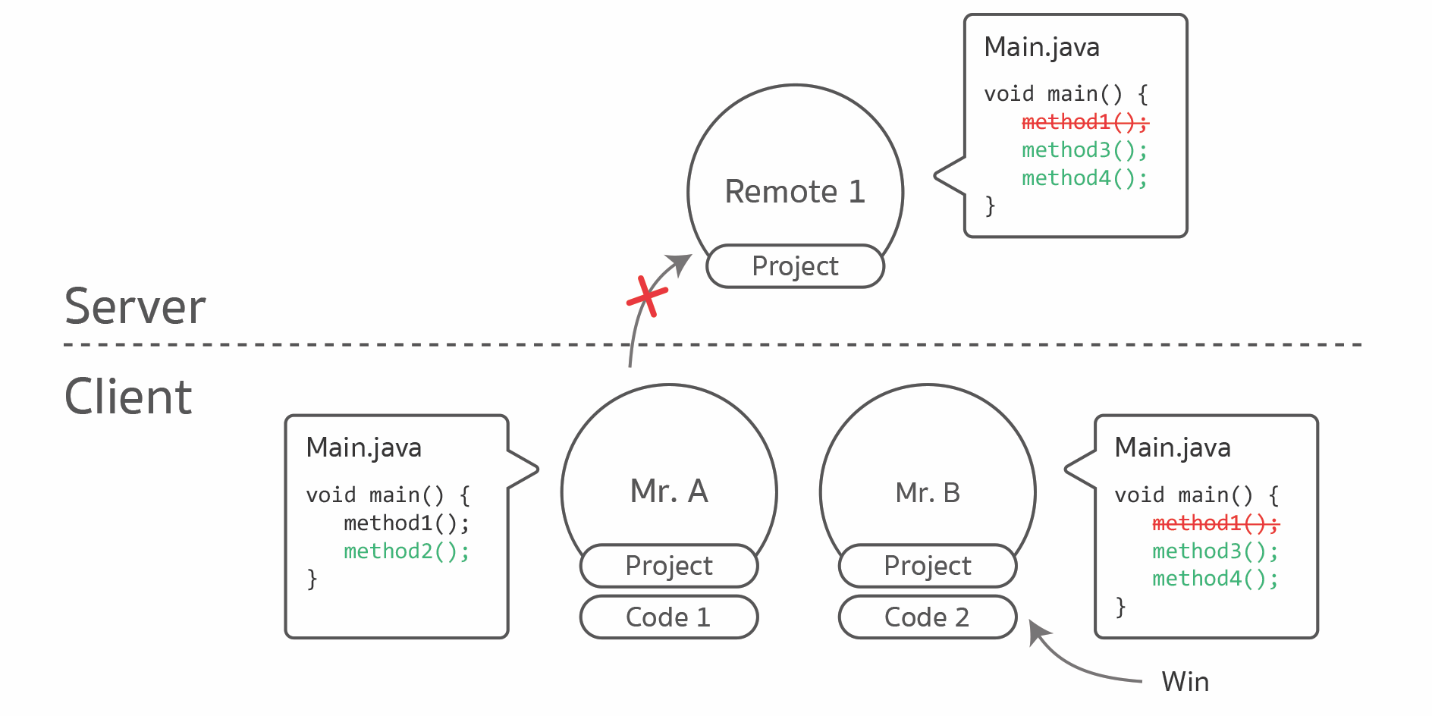
ສົມໝຸດ​​ວ່າ​ ທ້າວ A ກັບ​ ທ້າວ B ຂຽນ​ໂຄ້​ດ​ນຳກັນ​ຢູ່ ແລະ ​ທັງ​ຄູ່​ກໍ​ຂຽນ​ໂຄ້​ດ​ທີ່​ຢູ່ໃນ​ໄຟ​ລ​​ດຽວ​ກັນ



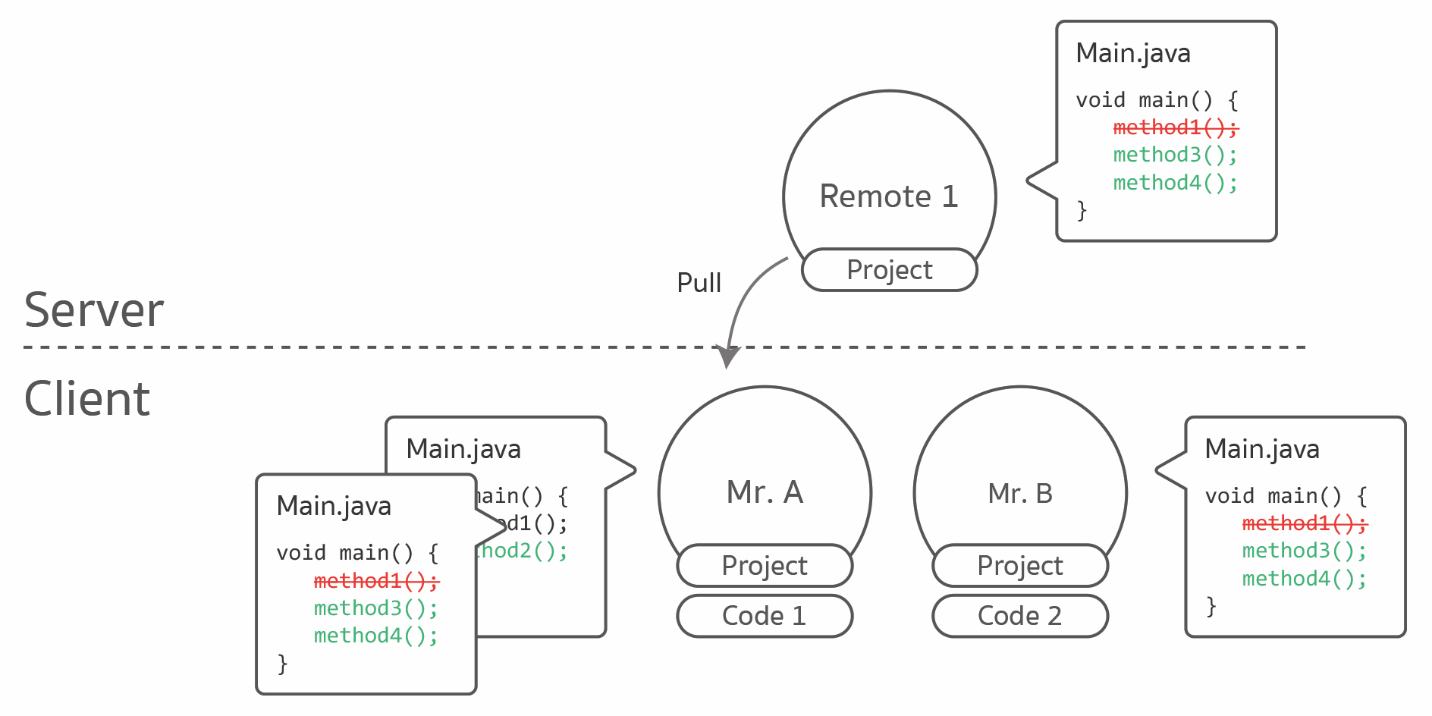
ທ້າວ B ຂຽນ​ໂຄ້​ດ​ສຳເລັດ​ແລ້ວ Commit ໄຟ​ລ​ທີ່​ຕົວ​ເອງ​ຂຽນ​ສຳເລັດ​ແລ້ວ Push ຂຶ້ນ Remote



ທ້າວ A ຂຽນ​ໂຄ້​ດ​ສຳເລັດ​ພາຍ​ຫລັງ ແລະ​ຈະ Push ຂຶ້ນ Remote ແຕ່ພົບວ່າ​ທ້າວ B ໄດ້ Push ຂຶ້ນ​ໄປ​ກ່ອນ​ແລ້ວ ຈຶ່ງ Push ຂຶ້ນ​ໄປ​ທັນ​ທີ​ບໍ່​ໄດ້



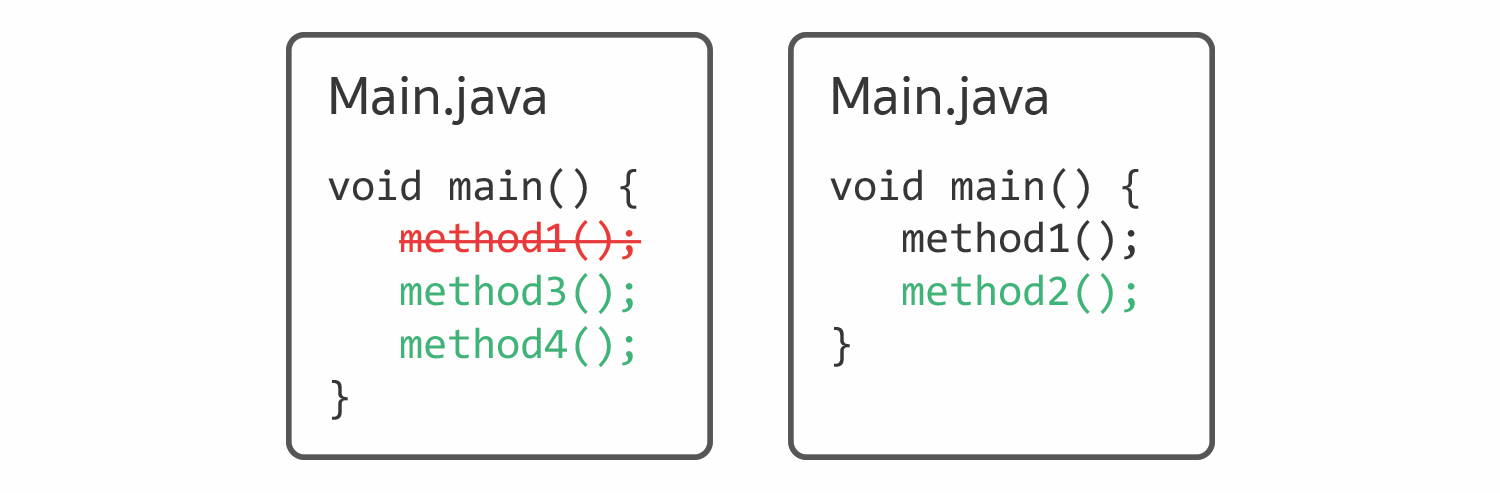
ດັ່ງ​ນັ້ນ​ສິ່ງ​ທີ່​ທ້າວ A ຕ້ອງ​ເຮັດ​ກ່ອນທີ່ຈະ Push ຂອງ​ຕົວ​ເອງ​ຂຶ້ນ​ໄປ​ໄດ້ກໍ​ຄື​ຈະ​ຕ້ອງ Pull ຈາກ Remote ລົງ​ມາ​ໃໝ່​ກ່ອນ​ເພື່ອ​ອັບ​ເດດ Commit ທີ່​ທ້າວ B ໄດ້ Push ຂຶ້ນ​ໄປ ຊຶ່ງ​ເຮົາ​ເອີ້ນຂັ້ນ​ຕອນ​ນີ້​ວ່າ Merge Commitນັ້ນ​ເອງ



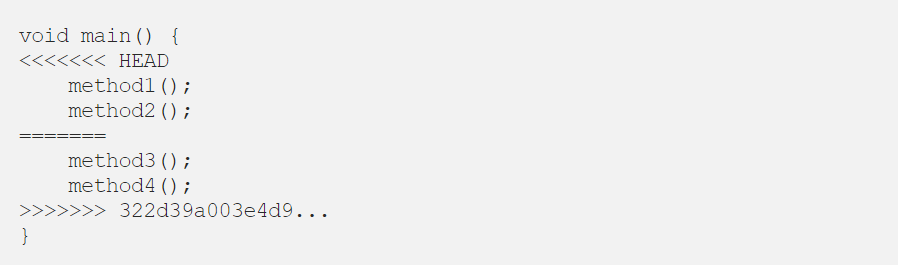
**ເພິ່ມເຕີມ** : ທີ່ຈິງແລ້ວ Git ສາມາດ Auto Merge ໄດ້​ເລຍ (ກົດ Pull > Merge ໄຟ​ລ​​ > Commit ໃຫ້​ທັນ​ທີ) ແຕ່​ກໍ​ແນະ​ນຳ​ວ່າ​ໃຫ້ Manual Merge ດີກວ່າ ເພື່ອ​ທີ່​ວ່າ​ຈະ​ໄດ້ Review Code ໄປ​ໃນ​ຕົວ​

## **Conflict**

ໃນ​ຂະນະ​ທີ່​ກຳ​ລັງ​ຈະ Merge ຢູ່​ນັ້ນຈະ​ເຫັນ​ວ່າ​ໂຄ້​ດ​ຂອງ​ທ້າວ A ແລະ​ ທ້າວ B ມັນ​ຕໍາ​ກັນ ໄປ​ແກ້​ໂຄ້​ດ​ຍ່ອນດຽວ​ກັນ ດັ່ງ​ນັ້ນ Git ຈະ​ແຈ້ງ​ວ່າ​ເກີດ Conflictຫລື ​ກໍ​ຄື​ໂຄ້​ດ​ທີ່​ທັບ​ຊ້ອນ​ກັນ​ນັ້ນ​ເອງ

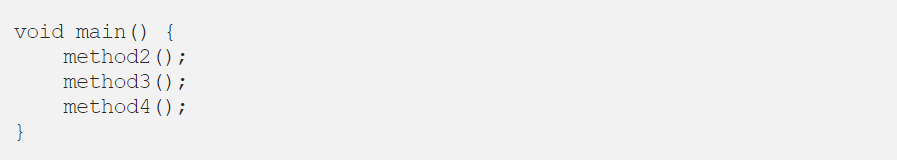


ຊຶ່ງ​ທ້າວ A ກໍ​ຕ້ອງ​ແກ້ Conflict ນີ້​ໃຫ້​ຮຽບຮ້ອຍ​ ເຖິງ​ຈະ Merge Commit ແລ້ວ Push ຂຶ້ນ Remote ໄດ້ ຊຶ່ງ​ຕອນ​ທີ່​ເກີດ Conflict ຂຶ້ນ ແລ້ວ​ເປີດ​ໂຄ້​ດ​ດັ່ງ​ກ່າວ​ມາ​ເບິ່ງ ກໍ​ຈະ​ເຫັນ​ເປັນ​ແບບ​ນີ້



ໃຫ້​ສັງເກດ​ຮູບ​ແບບ​ຂອງ​ເຄື່ອງ​ໝາຍ <<<, === ແລະ >>> ໃຫ້​ດີ​ໆ ແລ້ວ​ຈະ​ເຫັນ​ວ່າທີ່ຈິງແລ້ວ ​ມັນ​ມີ​ຮູບ​ແບບ​ທີ່​ເຂົ້າໃຈ​ໄດ້​ງ່າຍ​ຫລາຍໂດຍ​ທີ່

* ໂຄ້​ດ​ທີ່​ຢູ່​ລະຫວ່າງ <<< ແລະ === ຄື​ໂຄ້​ດ​ຂອງ​ທ້າວ A
* ໂຄ້​ດ​ທີ່​ຢູ່​ລະຫວ່າງ === ແລະ >>> ຄື​ໂຄ້​ດ​ຂອງ​ທ້າວ B ສ່ວນ​ຕົວ​ເລກ​ຕໍ່​ທ້າຍ​ຄື​ໝາຍ​ເລກ​ຂອງ Commit ບ່ອນ​ເຮັດວຽກ Merge

ເພື່ອ​ໃຫ້​ໂຄ້​ດ​ເຮັດວຽກ​ໄດ້​ເໝາະ​ສົມ ທ້າວ A ກໍ​ຈະ​ຕ້ອງ​ເອົາ​ໂຄ້​ດ​ທີ່​ຕົວ​ເອງ​ແກ້​ໄຂ ໄປ​ທັງຫມົດ​ກັບ​ໂຄ້​ດ​ຂອງ​ທ້າວ B ໃຫ້​ເຮັດວຽກ​ໄດ້

ເມື່ອ​ສຳເລັດ​ແລ້ວ​ກໍ​ໃຫ້ Commit (ຊຶ່ງ​ຈະ​ເປັນ​ການ Merge Commit) ແລ້ວ​ທ້າວ A ກໍ​ຈະ​ສາມາດ Push ຂຶ້ນ Remote ໄດ້​ແລ້ວ (ທ້າວ B ກໍ​ຈະ​ຕ້ອງ Pull Commit ພາຍ​ຫລັງ ແລະ ​ຖ້າ​ຫຼົງ​ໄປ​ແກ້​ໄຂ​​ຊ້ຳ​ກໍ​ອາດຈະ​ເກີດ Conflict ໄດ້)

Conflict ຖື​ວ່າ​ເປັນ​ເຫດການ​ທີ່ເກິດຂື້ນ​ໄດ້​ເປັນ​ປົກກະຕິ ໂດຍ​ສະເພາະ​ຢ່າງ​ຍິ່ງ​ໂປຣ​ເຈັກ​ໃຫຍ່​ໆ​ທີ່​ມີ Developer ຫຼາຍ​ໆ​ຄົນ​ຊ່ວຍ​ກັນ​​ຂຽນ​ໂຄ້​ດ ດັ່ງ​ນັ້ນ​, ການ​ແກ້ Conflict ຈຶ່ງ​ເປັນ​ໜຶ່ງ​ໃນ​ພື້ນ​ຖານ​ຂອງ​ການ​ໃຊ້​ງານ Git ທີ່​ນັກ​ພັດທະນາ​ຕ້ອງ​ເຂົ້າໃຈ​ ແລະ​ ຈັດການ​ກັບ​ມັນ​ບໍ່ໄດ້ ​ສະ​ນັ້ນ ​ຈະ​ເກີດ​ບັນຫາ​ຢ່າງ ​ເຊັ່ນ ຫຼົງໄປ​ລົບ​ໂຄ້​ດ​ຂອງ​ໝູ່ ໂດຍ​ບໍ່​ສົນ​ໃຈ​ຫຍັງເພື່ອ​ໃຫ້ Conflict ຫາຍ​ໄປເປັນ​ຕົ້ນ

* ເມື່ອ​ສະຫລຸບ​ຂັ້ນ​ຕອນ​ຂອງ​ການ​ແກ້ Conflict ຈະ​ເປັນ​ແບບ​ນີ້
* Pull ຈາກ Remote ລົງ​ເຄື່ອງ
* Conflict ເກີດ​ຂຶ້ນ
* ແກ້​ໄຂ​ໂຄ້​ດ​ໃຫ້​ເໝາະ​ສົມ
* Merge Commit
* Push ຂຶ້ນ Remote

## ສະຫລຸບ​ຮອບວຽຮ​ຂອງ Git ໃນ​ຊຸດທຳອິດ

* ເວລາ​ພັດທະນາ​ໂປຣ​ແກຣມກໍ​ຈະ​ມີ​ຂັ້ນ​ຕອນ​ແບບ​ນີ້
* ຂຽນ​ໂຄ້​ດ
* ເລືອກ​ໄຟ​ລ​ທີ່​ຕ້ອງ​ການ​ເຂົ້າ Staged
* Commit
* ຂຽນ​ໂຄ້​ດ​ຕໍ່
* ເລືອກ​ໄຟ​ລ​​ທີ່​ຕ້ອງ​ການ​ເຂົ້າ Staged
* Commit
* ວົນຫ​ລູບ​ໄປ​ເລື້ອຍ​ໆ​ຈົນ​ກວ່າ​ງານ​ຈະ​ສຳເລັດ
* ເວລາ​ຈະ Sync ຂໍ້​ມູນ​ໄປ​ທີ່ Remote ກໍ​ຈະ​ມີ​ຂັ້ນ​ຕອນ​ແບບ​ນີ້
* Fetch ເພື່ອ​ເຊັກ​ວ່າ​ມີ​ໃຜ Push ຫຍັງ​ຂຶ້ນ Remote ຫລື​ ບໍ່ (ບໍ່​ຈຳ​ເປັນ​ຕ້ອງ​ເຮັດ​ກໍ​ໄດ້)
* ຖ້າ​ມີ​ກໍ​ຈະ​ຕ້ອງ Pull ໂຄ້​ດ​ກ່ອນແລ້ວ Merge Commit ໃຫ້​ຮຽບຮ້ອຍ
* Push ຂໍ້​ມູນ​ຂຶ້ນ Remote
* ເວລາ​ຈະ Sync ຂໍ້​ມູນ​ໄປ​ທີ່ Remote ແບບ​ມີ Conflict ກໍ​ຈະ​ມີ​ຂັ້ນ​ຕອນ​ແບບ​ນີ້
* Fetch ເພື່ອ​ເຊັກ​ວ່າ​ມີ​ໃຜ Push ຫຍັງ​ຂຶ້ນ Remote ຫລື​ ບໍ່ (ບໍ່​ຈຳ​ເປັນ​ຕ້ອງ​ເຮັດ​ກໍ​ໄດ້)
* Pull ເພື່ອ​ດຶງ​ຂໍ້​ມູນ​ມາ​ໄວ້​ໃນ​ເຄື່ອງ
* ເກີດ Conflict
* ແກ້​ໄຂ​ໂຄ້​ດ​ໃຫ້​ເໝາະ​ສົມ
* Merge Commit
* Push ຂໍ້​ມູນ​ຂຶ້ນ Remote

# ຂຽນ​ໂຄ້​ດ​ໄປ​ຈົນ​ເຖິງ​ເມື່ອໃດ​ເຖິງ​ຈະ​ຕ້ອງ Commit?

ອັນ​ນີ້​ໜ້າ​ຈະ​ເປັນ​ຄຳ​ຖາມ​ທີ່ເກິດຂື້ນ​ເລື້ອຍ​ກັບ​ນັກ​ພັດທະນາ​ມື​ໃໝ່​ທີ່​ກຳ​ລັງ​ເລີ່ມ​ຕົ້ນ​ຮຽນ​ຮູ້​ ແລະ ​ຫັດ​ໃຊ້​ງານ Git ກັບ​ໂປຣ​ເຈັກ​ຂອງ​ຕົວ​ເອງ

ໃນ​ການ Commit ແຕ່​ລະ​ເທື່ອ​ເຖິງ​ແມ່ນ​ວ່າ​ຈະ​ບໍ່​ໄດ້​ມີ​ໃຜ​ມາ​ກຳນົດ​ມາດຕະຖານ ແຕ່​ການ Commit ທີ່​ດີ​ຄວນ​ເຮັດໃຫ້​ສາມາດ​ໄລ່​ໂຄ້​ດ​ພາຍ​ຫລັງ​ໄດ້​ງ່າຍ ສາມາດ​ເຮັດວຽກ​ຮ່ວມ​ກັບ​ຄົນ​ອື່ນ​ໄດ້ ແລະ​ຕ້ອງ​ເຮັດ​ຄືເກົ່າ​ທຸກ​ເທື່ອ​ຈົນ​ເປັນ​ນິ​ໄສ ບໍ່​ແມ່ນ​ວ່າຄິດ​ອອກ​ເມື່ອໃດ​ກໍ​ຄ່ອຍ Commit

ສຳລັບການ Commit ທີ່​ດີ​ຕ້ອງ​ກະ​ຊັບ ​ແລະ ​ນ້ອຍ​ຫລາຍ​ພໍ​ທີ່​ຈະ​ໄລ່​ເບິ່ງ​ໃນ History ໄດ້​ງ່າຍໆ ເວລາ​ມີ​ໂປຣ​ເຈັກ​​ໜຶ່ງ ຈະ​ຕ້ອງແຍກ​ສິ່ງ​ທີ່​ຕ້ອງ​ເຮັດ​​ອອກ​ມາ​ເປັນ Feature ແລ້ວ​ກໍ​ແບ່ງ​ຍ່ອຍ​ລົງ​ໄປ​ອີກ​ໃຫ້ເປັນ​ລະ​ດັບ Function ຈຶ່ງຈະ​ເລີ່ມ​ຂຽນ​ໂຄ້​ດ

ເມື່ອ​ເຮັດ​ໃນ​ແຕ່​ລະ Function ສຳເລັດ​ກໍ​ຈະ Commit ໃຫ້​ຮຽບຮ້ອຍແລ້ວ​ກໍ​ເຮັດ Function ຕໍ່ໄປ ແລະ​ ເຮັດວົນ​ແບບ​ນີ້​ໄປ​ເລື້ອຍ​ໆຈົນ Feature ນັ້ນ​ສຳເລັດ

ບໍ່​ແນະ​ນຳ​ໃຫ້​ເຮັດ​ໃນ​ລະ​ດັບ Feature ສຳເລັດ​ແລ້ວ​ຄ່ອຍ Commit ເພະ​ການເຮັດ​ແບບ​ນັ້ນ​​ຈະ​ໃຊ້​ປະໂຫຍດ Git ໄດ້​ແຕ່​ການ Backup ເທົ່າ​ນັ້ນ ແຕ່​ຈະ​ໄລ່​ເບິ່ງ History ເວລາ​ແກ້​ໄຂ​ໂຄ້​ດ​ໄດ້​ຍາກ ແລະ​ບໍ່​ສາມາດ​ຍ້ອນ​ຂໍ້​ມູນ​ກັບ​ໄດ້​ເລຍ ເພາະ​ມັນ​ຈະ​ຍ້ອນ​ກັບ​ໄປ​ທັງ​ໝົດ​ເລຍ ໂຄ້​ດ​ທີ່​ຂຽນ​ໄວ້​ທັງ​ໝົດ​ກໍ​ຈະ​ຫາຍ​ໄປ​ພ້ອມ​ໆ​ກັນ

ແລະ ​ກໍ​ບໍ່​ແນະ​ນຳ​ໃຫ້ Commit ໂຄ້​ດ​ທີ່​ຍັງ​ຂຽນ​ບໍ່​ສຳເລັດ​ເຊັ່ນ​ກັນ ໂຄ້​ດ​ທີ່​ຂຽນ​ບໍ່​ສຳເລັດ ​ໝາຍ​ເຖິງ​ໂຄ້​ດ​ທີ່​ບໍ່​ສາມາດ​ກົດ Run ເພື່ອ​ທົດສອບ​ໂປຣ​ແກຣມ​ໄດ້ ເພາະ​ຖ້າ​ເຮັດວຽກ​ຮ່ວມ​ກັບ​ຄົນ​ອື່ນ ແລ້ວ​ຄົນ​ອື່ນ Pull ຂໍ້​ມູນ​ໄປ ກາຍ​ເປັນ​ວ່າ​ຄົນ​ໆ​ນັ້ນ​ກໍ Run ໂປຣ​ແກຣມ​ບໍ່​ໄດ້​ເຊັ່ນ​ກັນ ຕ້ອງ​ເສຍ​ເວລາ​ນັ່ງ​ແກ້​ໂຄ້​ດ​ຂອງ​ເຮົາ​ອີກ

ແລະ​ ຢ່າ​ລືມ​ໃສ່​ໃຈ​ກັບ Commit Message ຫ້າມໃສ່​ຫຍັງ​ແບບ​ນີ້​ໂດຍເດັດ​ຂາດ ເພາະ​ມັນ​ຈະ​ບໍ່​ມີ​ທາງ​ຮູ້​ເລຍ​ວ່າ Commit ນັ້ນ​ເຮັດ​ຫຍັງ​ລົງ​ໄປ

* “Edit code”
* “Commit”
* “Refactor”
* **ຕ້ອງ Push ເມື່ອໃດ?**

ນອກ​ເໜືອ​ຈາກ​ການ Commit ແລ້ວການ Push ກໍ​ເປັນ​ອີກ​ຢ່າງ​ທີ່​ຫຼາຍ​ໆ​ຄົນ​ສົງ​ໄສ​ເຊັ່ນ​ກັນ. ເນື່ອງ​ຈາກ​ການ Push ນັ້ນໃນ​ບາງເທື່ອ​ກໍ​ຕ້ອງ Pull ແລ້ວ Merge Commit ໃຫ້​ຮຽບຮ້ອຍ​ກ່ອນ​ເຖິງ​ຈະ​ເຮັດ​ໄດ້ ດັ່ງ​ນັ້ນ​ບໍ່​ແນະ​ນຳ​ໃຫ້ Push ກ່ອນ​ປິດ​ຄອມ​ ເນື່ອງ​ຈາກ​ຖ້າ​ມີ​ຄົນ Push ກ່ອນ​ໜ້າ ແລ້ວ​ພໍ​ເຈົ້າ Pull ມາ​ກໍ​ພົ​ບວ່າ​ມັນ Conflict ສະນັ້ນ ສຸດ​ທ້າຍ​ແລ້ວ​ເຈົ້າ​ກໍ​ຕ້ອງ​ນັ່ງ​ແກ້ Conflict ຊຸດ​ໃຫຍ່​ ດັ່ງ​ນັ້ນ​ທາງ​ທີ່​ດີ​ຄວນ Push ທຸກ​ເທື່ອ​ທີ່​ມີ​ໂອ​ກາດ​ດີກວ່າ ຢ່າງ​ນ້ອຍ​ຖ້າ​ມັນ​ຈະ Conflict ເຈົ້າ​ກໍ​ໄດ້​ແກ້​ຕັ້ງ​ແຕ່​ເນີ້ນ​ໆ ​ແລະ ​ບໍ່​ຫລາຍປານໃດ ທີ່​ສຳຄັນ​ມັນ​ຈະ​ຊ່ວຍ​ໃຫ້​ທຸກ​ຄົນ​ໃນ​ທີມ​ຮູ້ Progress ນຳກັນ​ໄດ້​ຈາກ​ສ​ະຖາ​ນະ​ການ Push ອີກດ້ວຍ ​ແລະ ​ນອກ​ຈາກ​ນີ້ ໃນ​ບາງ​ບາອນ​ທີ່​ມີ​ການເຮັດ Continuous Integration (CI) ທີ່​ຈະ​ດຶງ Commit ຈາກ Remote ໄປ​ເຮັດ Testing ຖ້າ​ເຮົາ Push ເລື້ອຍໆ​ແລ້ວ Test ເກີດ Failed ຂຶ້ນ​ມາ​ກໍ​ຈະ​ເຮັດໃຫ້​ຮູ້​ຕົວ​ໄດ້​ໄວ ​ແລະ​ ແກ້​ໄຂ​ໄດ້​ທັນ​ທີ

# ຄຳ​ຕ່າງ​ໆ​ທີ່​ຈະ​ຕ້ອງ​ຮູ້​ຈັກ​ເມື່ອ​ໃຊ້​ງານ Git (ຊຸດ​ທີ່ 2)

ເມື່ອ​​ຮູ້​ຈັກ​ຄຳ​ຕ່າງ​ໆ​ຂອງ Git ໃນ​ຊຸດ​ທີ່ 1 ແລ້ວນັ້ນ​ໝາຍ​ຄວາມ​ວ່າ​ເຮົາ​ສາມາດ​ໃຊ້​ງານ Git ເບື້ອງ​ຕົ້ນ​ໄດ້​ແລ້ວ ແຕ່​ນັ້ນ​ກໍ​ຍັງ​ບໍ່​ພໍ ເພາະ​ຄຳ​ຕໍ່ໄປ​ນີ້​ຄື​ສິ່ງ​ທີ່​ຄວນ​ຈະ​ຮູ້​ຈັກ​ ແລະ ​ເຂົ້າໃຈ​ນຳກັນ​ເຊັ່ນ​ກັນ

* BranchPull Request
* Merge Branch
* Stash
* Unstash
* Git Flow
* Tag
* Fork
* Pull Request

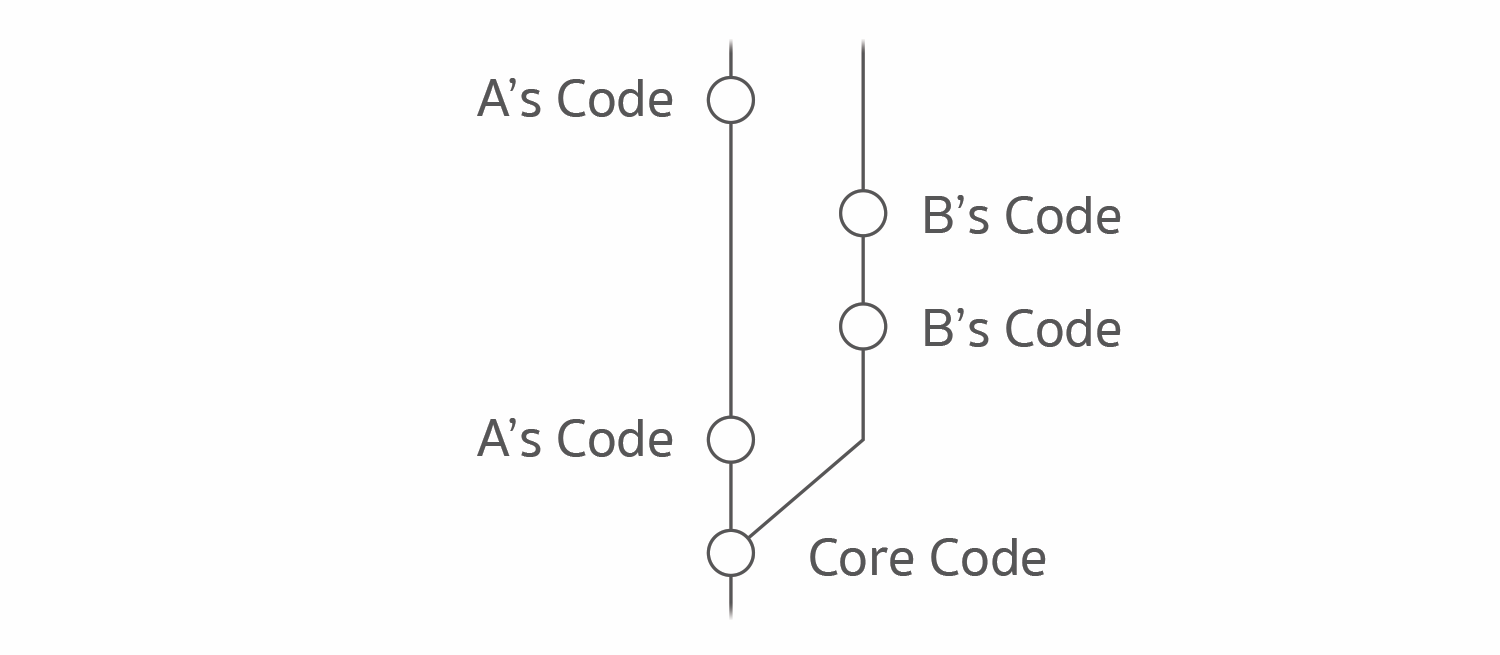
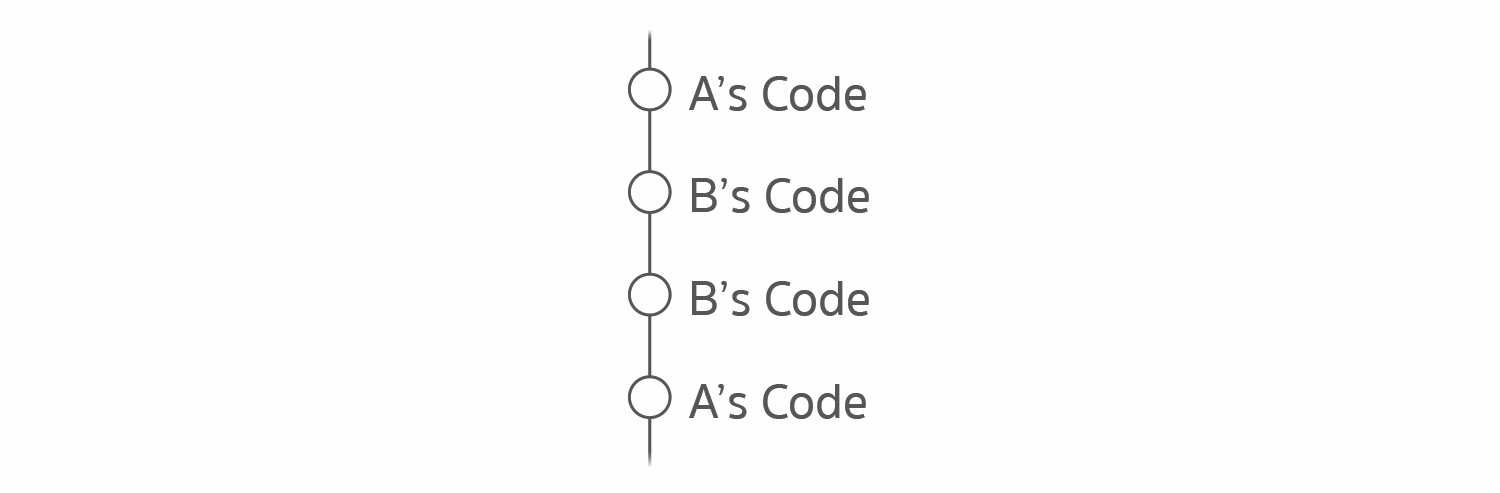
## **Branch**

ເວລາ​ທີ່​ເຮົາ Commit ເລື້ອຍ​ໆ​ຈົນ​ກາຍ​ເປັນ History ມັນ​ກໍ​ຈະ​ມີ​ໜ້າ​ຕາ​ແບບ​ນີ້

ການ​ລຽງ​ກັນ​ຂອງ Commit ຈົນ​ເປັນ​ເສັ້ນ​ຍາວ​ໆ​ແບບ​ນີ້​ຈະ​ເອີ້ນວ່າ Branchແລະ​ ໃນ​ການ​ໃຊ້​ງານ Git ແທ້ໆ​ຈະ​ບໍ່​ໄດ້​ມີ​ແຕ່ Branch ດຽວ​ເທົ່າ​ນັ້ນ​

ຈະ​ເກີດ​ຫຍັງ​ຂຶ້ນ​ຖ້າທ້າວ A ແລະ​ ທ້າວ B ເຮັດ​ໂປຣ​ເຈັກ​ດຽວ​ກັນແຕ່​ວ່າ​ເຮັດ​ຄົນ​ລະ Feature?

ຖ້າ​ເຮົາ​ເຮັດວຽກ​ກັນ​ແບບ Branch ດຽວເຮົາ​ກໍ​ຈະ​ພົ​ບວ່າ​ທ້າວ A ກັບ​ທ້າວ B ຕ້ອງ​ເສຍ​ເວລາ Pull ຂໍ້​ມູນ​ຂອງ​ກັນ ​ແລະ ​ກັນ​ເລື້ອຍ​ຫລາຍທັງ​ໆ​ທີ່​ໂຄ້​ດ​ຂອງ​ທັງ​ສອງ​ບໍ່​ໄດ້​ກ່ຽວ​ຂ້ອງ​ກັນ​

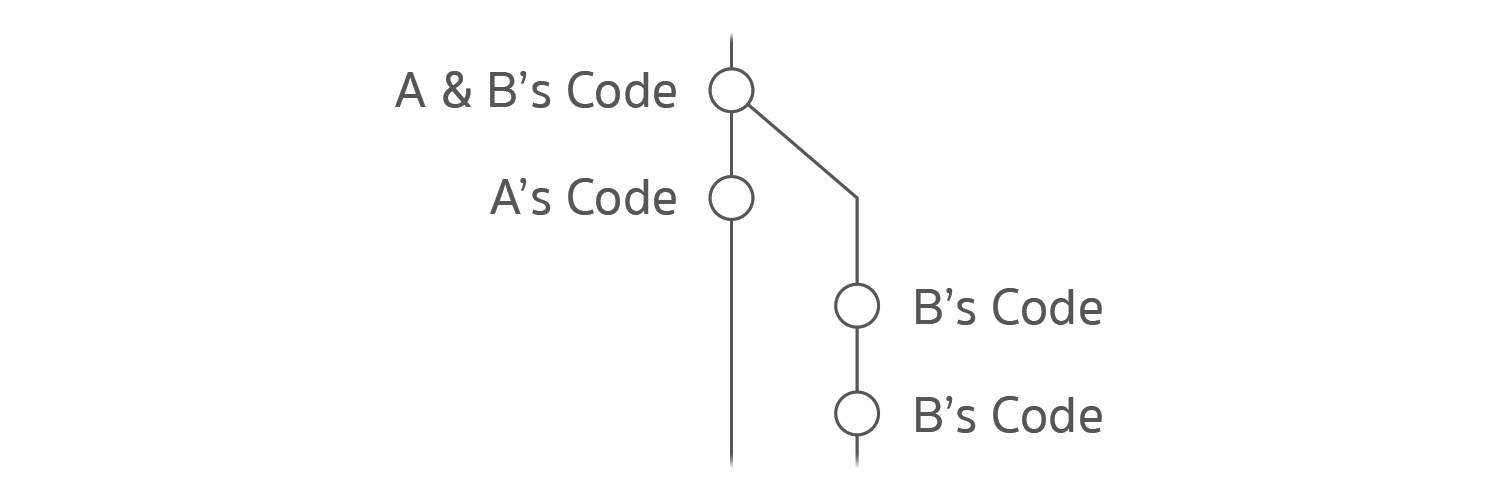
ຊຶ່ງ Git ອອກ​ແບບ​ມາ​ໃຫ້​ສ້າງ Branch ແຍກ​ອອກ​ມາ​ໄດ້​ຫຼາຍ​ເສັ້ນ ດັ່ງ​ນັ້ນ​ ໃນ​ກໍລະນີ​ດັ່ງ​ກ່າວ​ເຮົາ​ຈຶ່ງ​ຄວນ​ແຍກ Branch ອອກ​ມາ​ສຳລັບ​ທ້າວ A ແລະ​ ອີກ Branch ສຳລັບ​ທ້າວ B

ພໍ​ທັງ​ສອງ​ຄົນ​ຂຽນ​ໂຄ້​ດ​ຂອງ​ຕັວ​ເອງ​ສຳເລັດ​ແລ້ວ​ກໍ​ເຖິງ​ເວລາ​ຂອງ

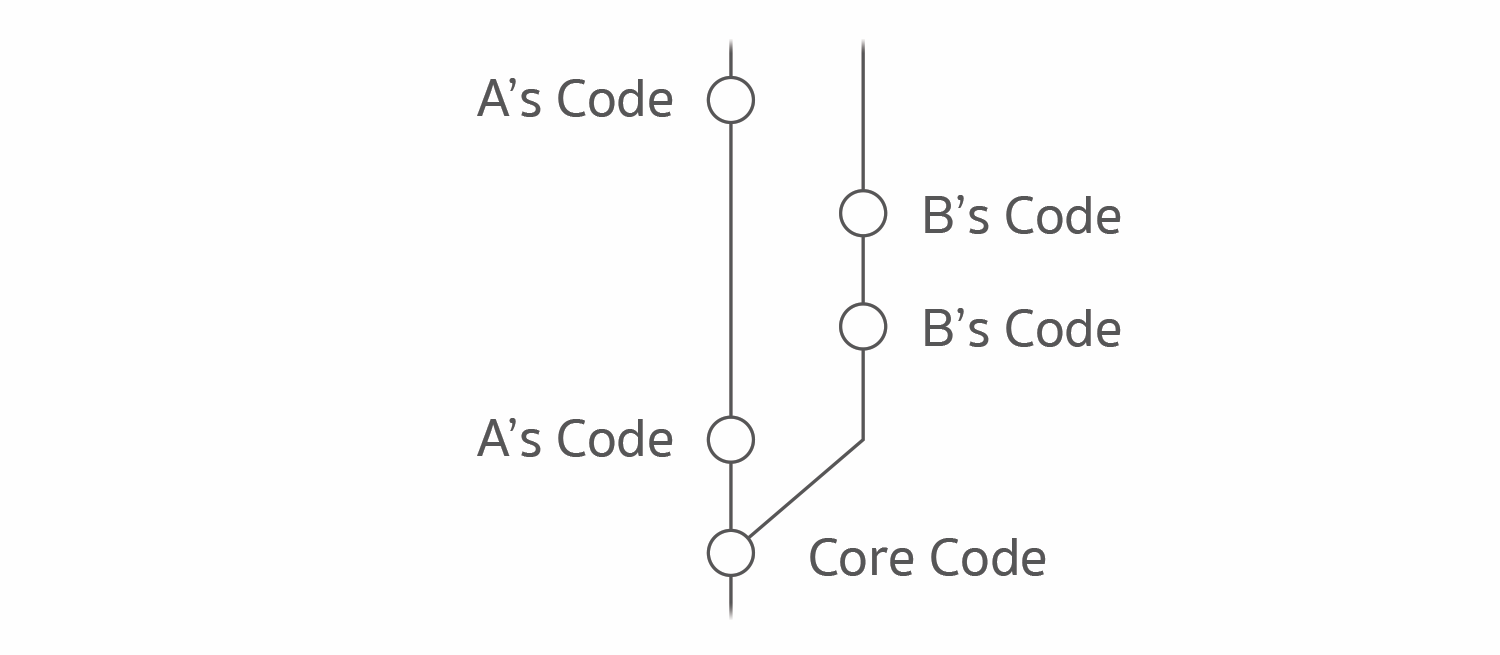
## **Merge Branch**

ເມື່ອ​ໃດ​ກໍ​ຕາມ​ທີ່​ຢາກ​ຈະ​ຈັບ Branch ຄູ່​ໃດ​ມາ Merge ລວມ​ກັນ ນີ້ແຫລະຄື Merge Branch

ແຕ່​ໃນ​ລະຫວ່າງ​ການ Merge Branch ກໍ​ອາດຈະ​ມີ Conflict ເກີດ​ຂຶ້ນ ແຕ່​ເນື່ອງ​ຈາກ​ການ​ແຍກ Branch ມັກ​ຈະ​ໃຊ້​ສຳລັບ​ກໍລະນີ​ທີ່​ຂຽນ​ໂຄ້​ດ​ຄົນ​ລະ​ສ່ວນ​ກັນ​ຢູ່​ແລ້ວ ດັ່ງ​ນັ້ນ ​ຕຳແໜ່ງ​ຂອງ​ໂຄ້​ດ​ທີ່​ເກີດ Conflict ກໍ​ຈະ​ມີ​ພຽງ​ບາງ​ຈຸດ​ເທົ່າ​ນັ້ນ ແລະ ​ເມື່ອ​ແກ້​ໄຂ Conflict ຈົນ​ສຳເລັດ​ແລ້ວ Merge Branch ທັງ​ສອງ​ເຂົ້າ​ນຳກັນ​ໄດ້​ແລ້ວ ມັນ​ກໍ​ຈະ​ໄດ້​ອອກ​ມາ​ເປັນ Merge Commit ນັ້ນ​ເອງ

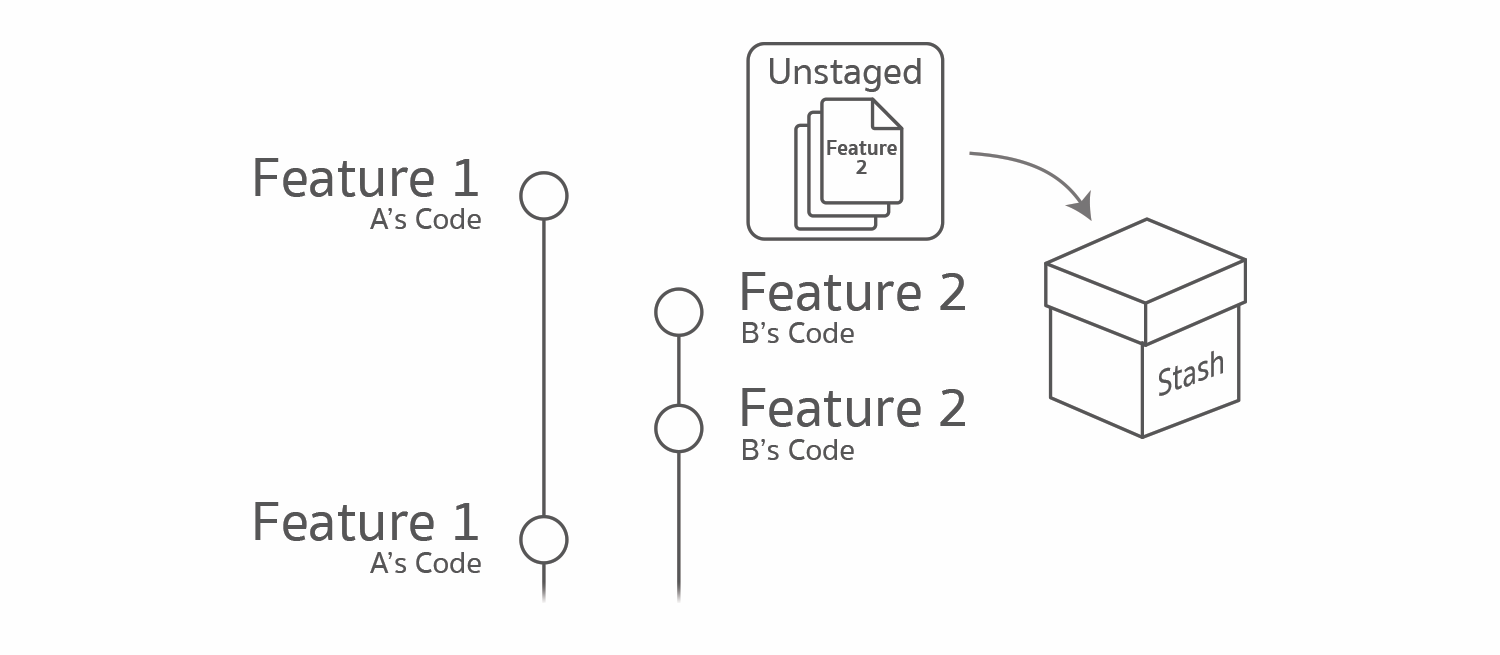


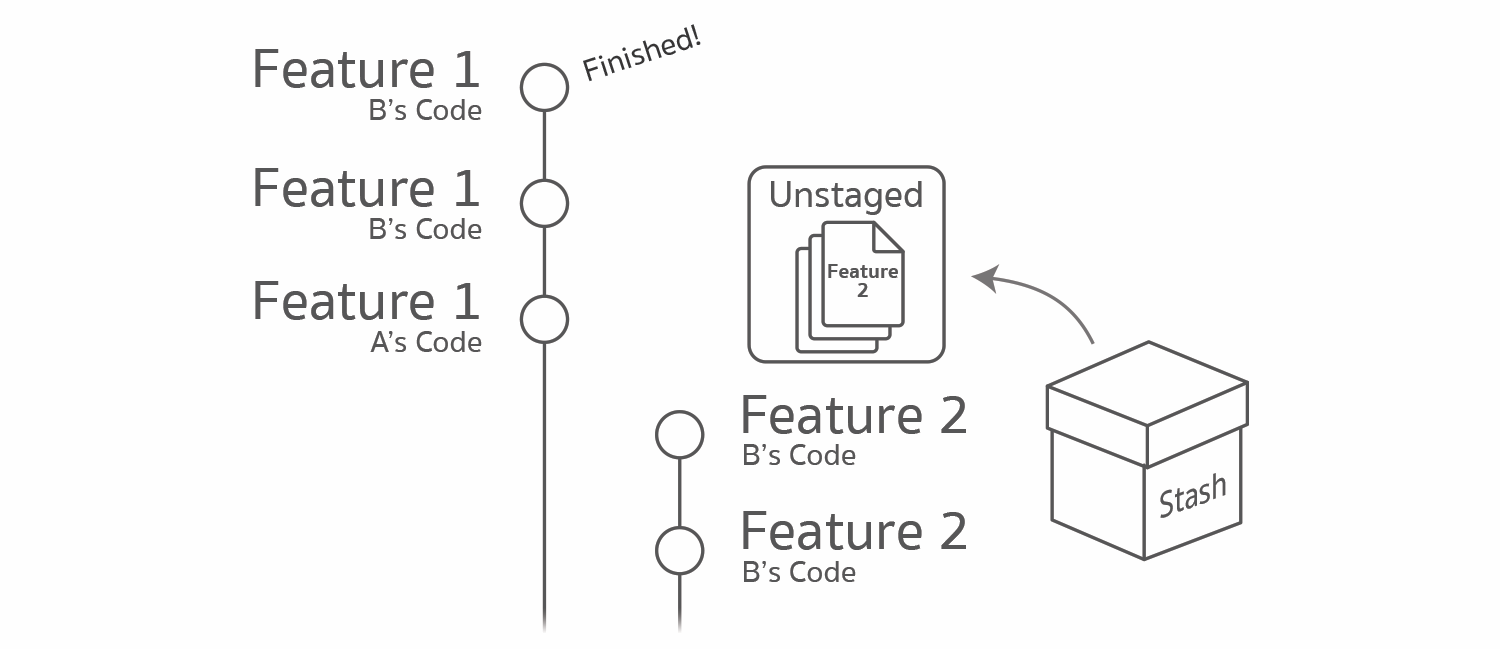
## **Stash ແລະ Unstash**

ສົມມຸດ​ວ່າ​ໂປຣ​ເຈັກ​ໜຶ່ງ​ຖືກ​ແຍກ​ອອກ​ມາ​ເປັນ 2 Branch ນຳກັນ ແລ້ວ​ກໍ​ເປັນ​ຫນ້າທີ່​ຂອງ​ທ້າວ A ແລະ​ ທ້າວ B ທີ່​ຈະ​ຕ້ອງ​ຮັບ​ຜິດຊອບ​ຕາມເຄີຍ

ແຕ່​ຢູ່​ດີ​ໆ​ກໍເກິດ​ເຫດ​ດ່ວນ ເພາະ​ທ້າວ A ​ຂຽນ​ໂຄ້​ດ​ບໍ່​ທັນແລ້ວ Feature ຂອງ​ທ້າວ A ຕ້ອງ​ສົ່ງ​ໃຫ້​ລູກ​ຄ້າ​ ຈຶ່ງ​ເຮັດໃຫ້​ທ້າວ B ທີ່​ຂຽນ​ໂຄ້​ດ​ໃນ Feature ຂອງ​ຕົວ​ເອງ​ຢູ່ ກໍ​ຕ້ອງ​ຢຸດ​​ກາງ​ຄັນ​ເພື່ອສະຫຼັບ​ໄປ​ຊ່ວຍ​ທ້າວ A ຂຽນ​ໃຫ້​ສຳເລັດ

ແຕ່​ ມີ​ບາງ​ໄຟ​ລ​​ທີ່​ທ້າວ B ຂຽນ​ຄ້າງ​ໄວ້​ຍັງ​ບໍ່​ສຳເລັດ ແລະ ຍັງ​ຈະ Commit ເລີຍ​ກໍ​ເຮັດ​ບໍ່​ໄດ້ ອີກ​​ຕ້ອງ​ສ​ະຫຼັບ​ໄປ Branch ຂອງ​ທ້າວ A ດ່ວນ​

ດັ່ງ​ນັ້ນ ​ທ້າວ B ຈະ​ຕ້ອງ​ເອົາ​ໄຟ​ລ​​ທີ່​ຢູ່ໃນ Unstaged ໄປ​ເກັບ​ໄວ້​ໃນ Stashຊົ່ວວ​ຄາວ​ກ່ອນ ຈະ​ໄດ້​ສ​ະຫຼັບ Branch ໄປ​ຊ່ວຍທ້າວ A ຂຽນ​ໂຄ້​ດ​ໃຫ້​ສຳເລັດ​ທັນ​ເວລາ

ເມື່ອທ້າວ B ໄປ​ຊ່ວຍ​ທ້າວ A ຂຽນ​ຈົນ​ສຳເລັດ​ແລ້ວ ກໍ​ກັບ​ມາ Branch ຂອງ​ຕົວ​ເອງ​ແລ້ວ Unstashເພື່ອ​ເອົາ​ໄຟ​ລ​​ທີ່​ເຄີຍ​ຂຽນ​ຄ້າງ​ໄວ້​ກັບ​ມາ​ເຮັດ​ຕໍ່​ໃຫ້​ສຳເລັດ. ​ນອກ​ຈາກ​ນີ້ Stash ຍັງ​ສາມາດ​ໃຊ້​ຢ້າຍ​ໄຟ​ລ​ຂ້າມ Branch ໄດ້​ອີກ​ດ້ວຍ​

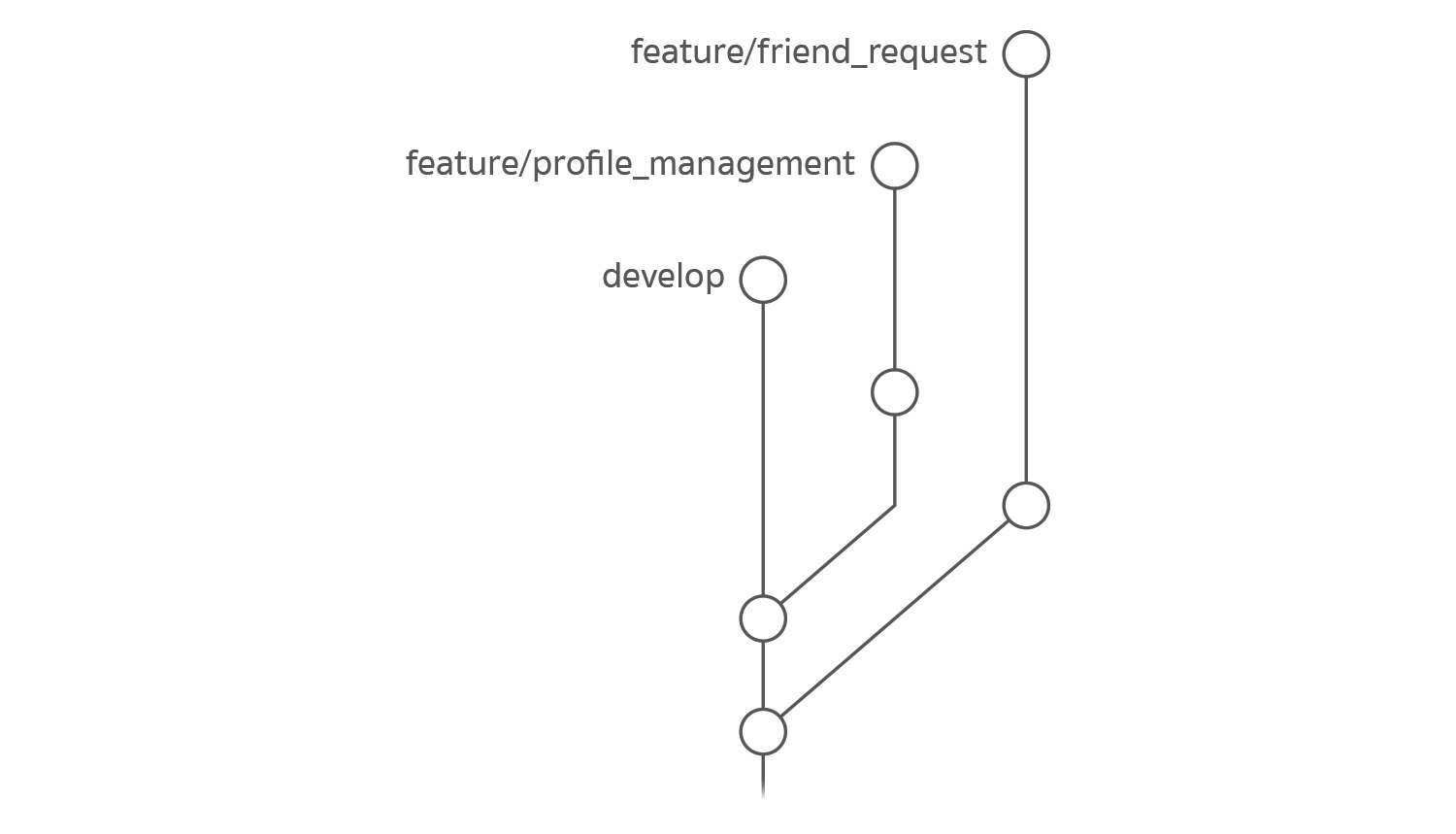
## **Git Flow**

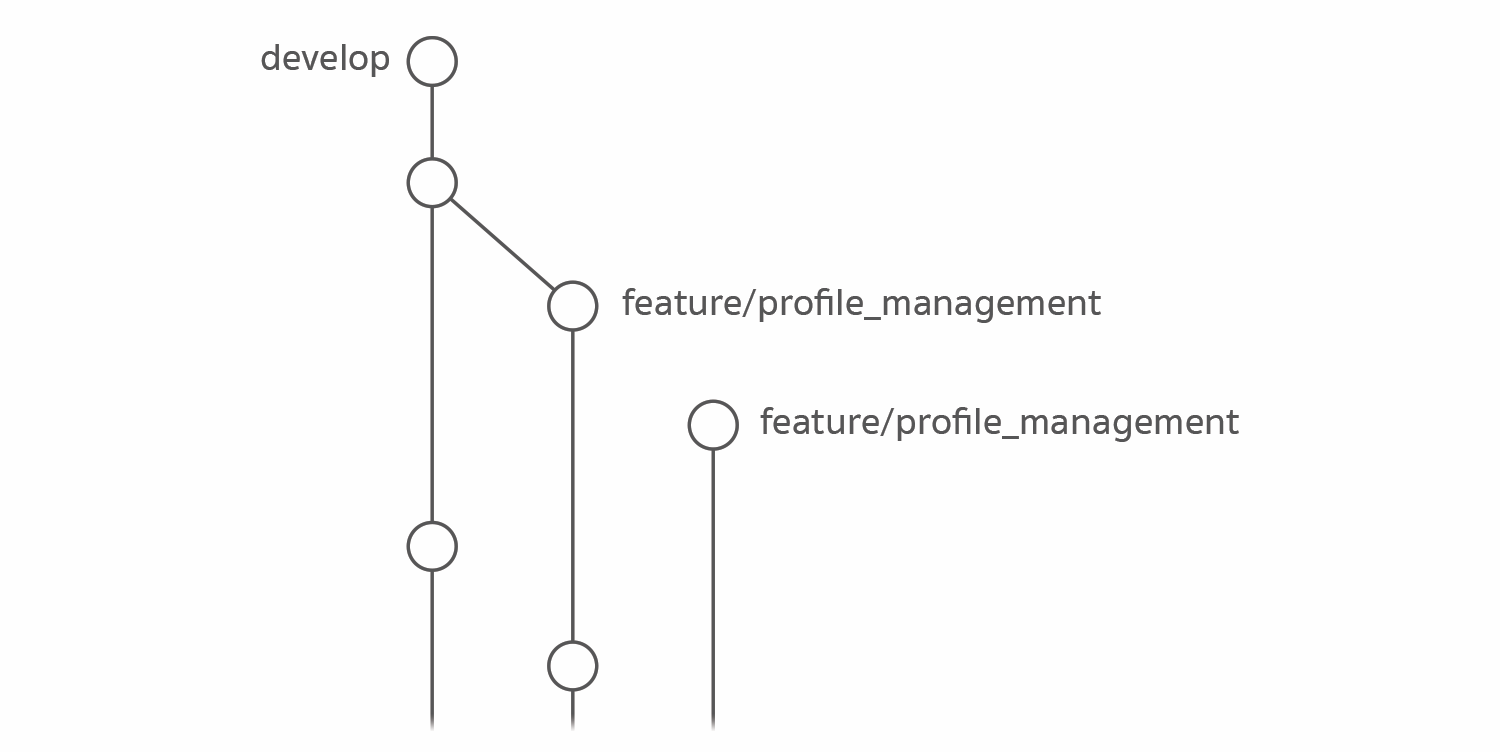
ໃນ Development Process ແທ້ໆ​ນັ້ນ​ມີ​ຫຍັງ​ອີກ​ຫຼາຍ​ໆ​ຢ່າງ ບໍ່​ວ່າ​ຈະ​ເປັນ​ການ​ພັດທະນາ​ຫຼາຍ​ໆ Feature ພ້ອມ​ໆ​ກັນແຕ່​ເວລາ​ຂຶ້ນ Production ຈະ​ເວົ້າສະເພາະ​ບາງ Feature ເທົ່າ​ນັ້ນສ່ວນ Feature ທີ່​ເຫຼືອ​ກໍ​ອາດຈະ​ພັດທະນາ​ຕໍ່​ ຫລື ​ປ່ອຍ​ຖິ້ມ​ໄວ້​ຖ້າຂຶ້ນ​ໃນ​ຮອບ​ໜ້າ ທັງຫມົດ​ໄປ​ເຖິງ​ຂັ້ນ​ຕອນ​ກ່ອນທີ່ຈະ​ຂຶ້ນ Production ທີ່​ຈະ​ຕ້ອງ​ໃຫ້ QA/Tester ທົດສອບ​ກ່ອນ ຫລື ​ການເຮັດ Hotfix ເພາະ​ເຈີ​​ບັກ​ໃນ Production ທີ່​ຕ້ອງ​ແກ້​ດ່ວນ

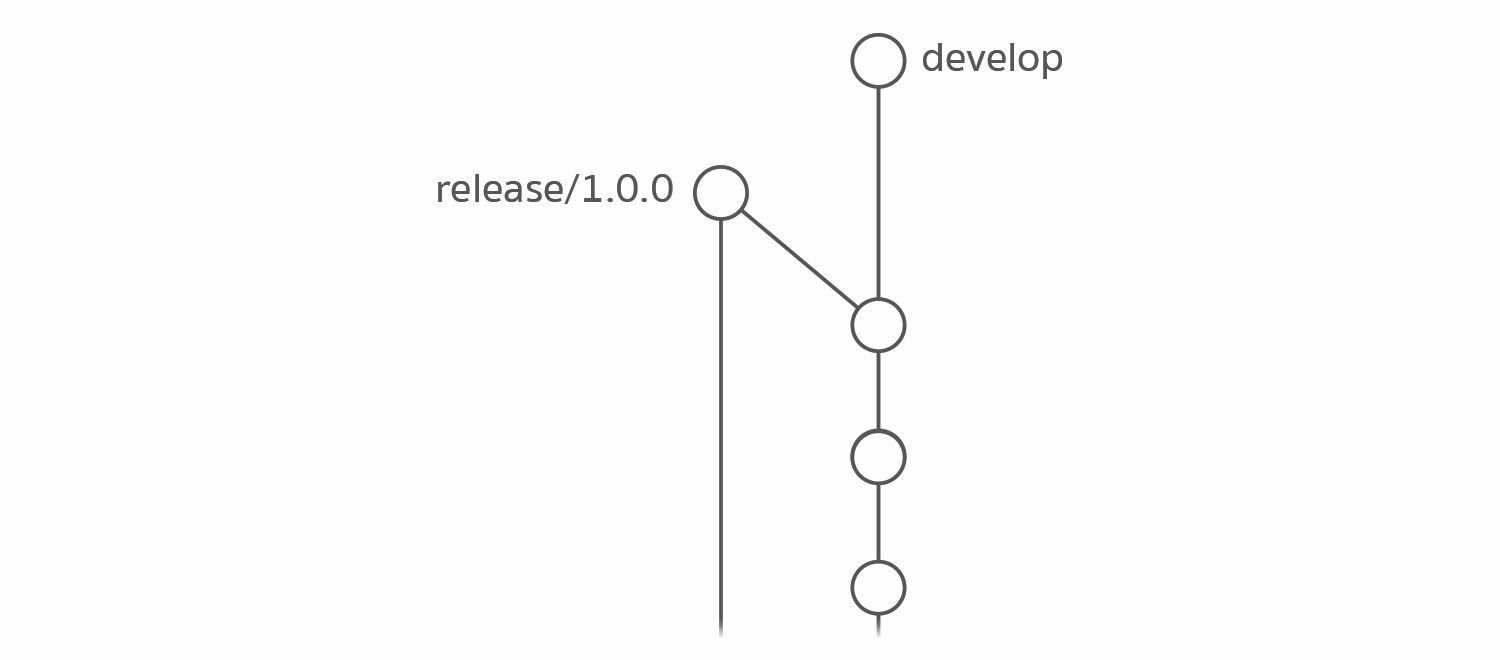
ດັ່ງ​ນັ້ນ​ເຮົາ​ຈຶ່ງ​ສາມາດ​ໃຊ້​ຄວາມ​ສາມາດ​ໃນ​ການ​ແຍກ Branch ຂອງ Git ມາ​ຊ່ວຍ​ແກ້​ບັນຫາ​ນີ້​ໄດ້ ແລະ ​ເມື່ອ​ວິທີ​ແບບ​ນີ້​ນິຍົມ​ຫລາຍ​ຂຶ້ນ​ເລື້ອຍ​ໆ​ກໍ​ເຮັດໃຫ້​ເກີດ​ສິ່ງ​ທີ່​ເອີ້ຍວ່າ Git Flowຂຶ້ນ​ມາ

Git Flow ຄື​ໜຶ່ງ​ໃນ​ມາດຕະຖານ​ທີ່​ກຳນົດ​ຮູບ​ແບບ​ຂອງ​ການ​ແຕກ Branch ໃຫ້ເປັນ​ລະບົບ​ລະບຽບ​ເພື່ອ​ຕອບ​ໂຈ​ດໃນ Development Process ໂດຍ​ແບ່ງ Branch ອອກ​ເປັນ 5 ກຸ່ມ​ໃຫຍ່​ໆ​ດັ່ງ​ນີ້

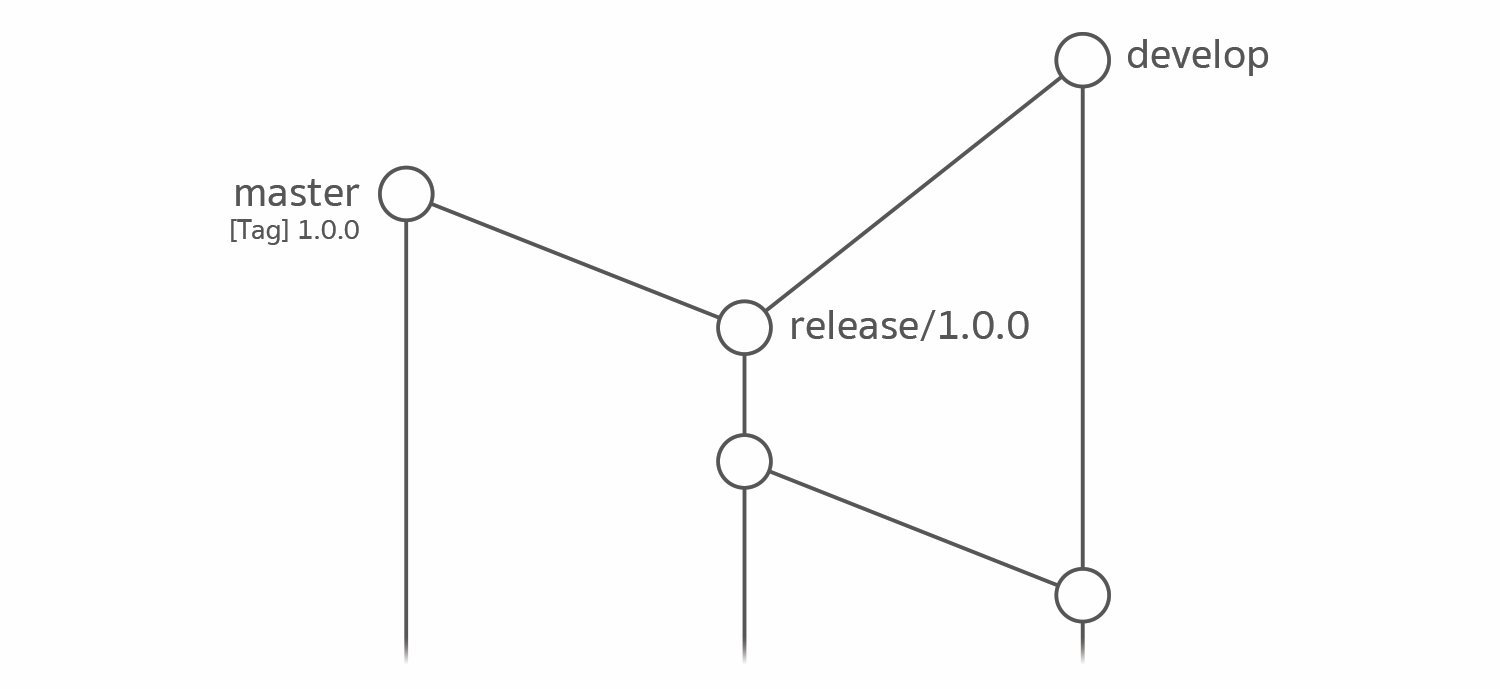
* master
* develop
* feature
* release
* hotfix

ຂັ້ນ​ຕອນ​ການ​ຂຽນ​ໂຄ້​ດ​ທັງ​ໝົດ​ຈະ​ຢູ່ໃນ developແລະ feature ຢູ່​ສະເໝີ ໂດຍ​ຕອນ​ຂຶ້ນ​ໂຄງ​ໂປຣ​ເຈັກ​ໃນ​ຕອນ​ເລີ່ມ​ຕົ້ນ​ອາດຈະ​ເຮັດ​ໃນ develop ແຕ່​ເມື່ອ​ໂຄ້​ດ​ພື້ນ​ຖານ​ພ້ອມ​ແລ້ວ ກໍ​ຈະ​ແຍກ Branch ຕາມ Feature ເພື່ອ​ຂຽນ​ໂຄ້​ດ ແລະ ​ນັ້ນ​ກໍ​ຄື branch ທີ່​ຊື່​ວ່າ feature ນັ້ນ​ເອງ

ເມື່ອ​ພັດທະນາ Feature ຕົວ​ໃດ​ສຳເລັດ ແລະ​ ພ້ອມ​ຈະ Release ກໍ​ຈະ Merge ເຂົ້າໄປ​ໃນ develop ເລີຍ ສ່ວນ​ຕົວ​ໃດ​ທີ່​ຍັງ​ບໍ່​ພ້ອມ Release ກໍ​​ຖິ້ມ​ໄວ້​ແບບ​ນັ້ນ​ແຫຼະ

ເມື່​ອເຖິງ​ເວລາ​ອັນ​ສົມຄວນ​ທີ່​ຈະ Release ເພື່ອ​ຕຽມ​ຂຶ້ນ Production ແລ້ວກໍ​ຈະ​ແຕກ Branch ອອກ​ມາ​ເປັນ releaseໂດຍ​ມີ Sub Branch ເປັນ​ຊື່​ເວີຊັນ​ທີ່​ຕ້ອງ​ການ

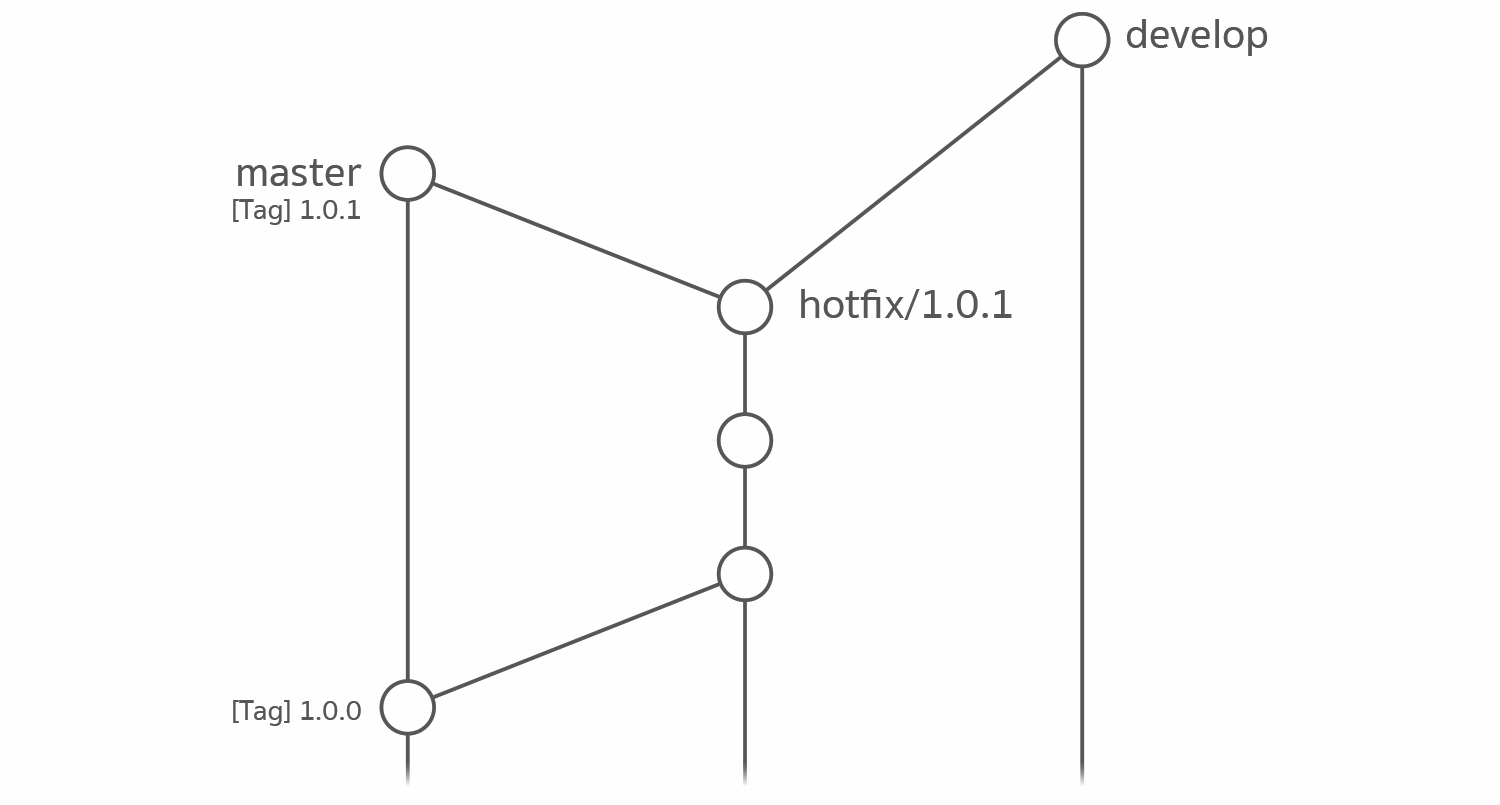
ຊຶ່ງ​ການ​ແຕກ​ເຂົ້າ​ສູ່ release ນີ້​ມີ​ໄວ້​ເພື່ອ​ໃຫ້ QA/Tester ກໍ່ການ​ທົດສອບ​ໂປຣ​ແກຣມ​ຂອງ​ເຮົາ​ໃຫ້​ຜ່ານ​ກ່ອນທີ່ຈະ​ຂຶ້ນ Production ນັ້ນ​ເອງ ດັ່ງ​ນັ້ນ ​ຖ້າ​ມີ Issue ເກີດ​ຂຶ້ນ​ລະຫວ່າງ​ນີ້​ກໍ​ຕ້ອງ​ແກ້​ໄຂ​ໃນ​ນີ້​ໃຫ້​ຮຽບຮ້ອຍ​

ເມື່ອ​ QA/Tester ຮຽບຮ້ອຍ​ແລ້ວ ກໍ​ເຖິງ​ເວລາ​ພ້ອມ​ທີ່​ຈະ​ເຂົ້າ​ສູ່ masterແລ້ວ​

ສຳລັບ​ການ Finish Branch ຂອງ release ຈະ​ຕ້ອງ Merge ເຂົ້າ master ແລະ develop ທຸກ​ເທື່ອ ສາ​ເຫດ​ທີ່​ຕ້ອງ Merge ເຂົ້າ develop ດ້ວຍ​ກໍ​ເພາະ​ວ່າ​ມັນ​ມີ​ໂຄ້​ດ​ທີ່​ແກ້ Issue ໃນ​ລະຫວ່າງ​ການ​ທົດສອບ​ຂອງ QA/Tester ເພີ່ມ​ເຂົ້າ​ມາ

ຈະ​ເຫັນ​ວ່າ​ທີ່ master ນັ້ນ​ຈະ​ເປັນ​ໂຄ້​ດ​ທີ່​ສົມບູນ​ທີ່ສຸດ (ຂອງ​ເວີຊັນ​ນັ້ນ​ໆ) ແລະ​ ຈະ​ບໍ່​ມີ​ການ Commit ເຂົ້າໄປ​ໂດຍຕົງ​ເລີຍ ແຕ່​ຈະ​ມາ​ຈາກ​ການ Merge ຂອງ release ເທົ່າ​ນັ້ນ ດັ່ງ​ນັ້ນ master ຈຶ່ງ​ມີ​ແຕ່​ໂຄ້​ດ​ທີ່​ສຳເລັດ​ສົມບູນ ເວລາ​ຈະ Rollback ແບບ​ກະ​ທັນ​ຫັນ​ກໍ​ສາມາດ​ເຮັດ​ໄດ້​ໂດຍ​ບໍ່​ຕ້ອງ​ກັງ​ວົນ​ວ່າ​ຈະ​ມີ​ບັນຫາ​ຫຍັງ​ ຫລື​ ບໍ່

ໃນ​ກໍລະນີ​ທີ່​ Production ​ລ່າ​ສຸດ​ເກີດ Critical Issue ຂຶ້ນ​ມາ ແຕ່​ບໍ່​ເຖິງ​ກັບ​ຕ້ອງ Rollback ກໍ​ຈະ​ມີ hotfixເພື່ອ​ເຂົ້າ​ມາ​ແກ້​ບັນຫາ​ນີ້​ໃຫ້​ໂດຍ​ສະເພາະ



ຫນ້າທີ່​ຂອງ hotfix ກໍ​ຕາມ​ຊື່​ຂອງ​ມັນ​ເລີຍ ມີ​ໄວ້​ແກ້​ບັ​ກ​ເທິງ Production ແບບ​ດ່ວນ​ ໂດຍ hotfix ຈະ​ມີ Sub Branch ເປັນ​ຊື່​ເວີຊັນ (ນິຍົມ​ປ່ຽນ​ສະເພາະ​ເລກ​ຂອງ Patch Version)

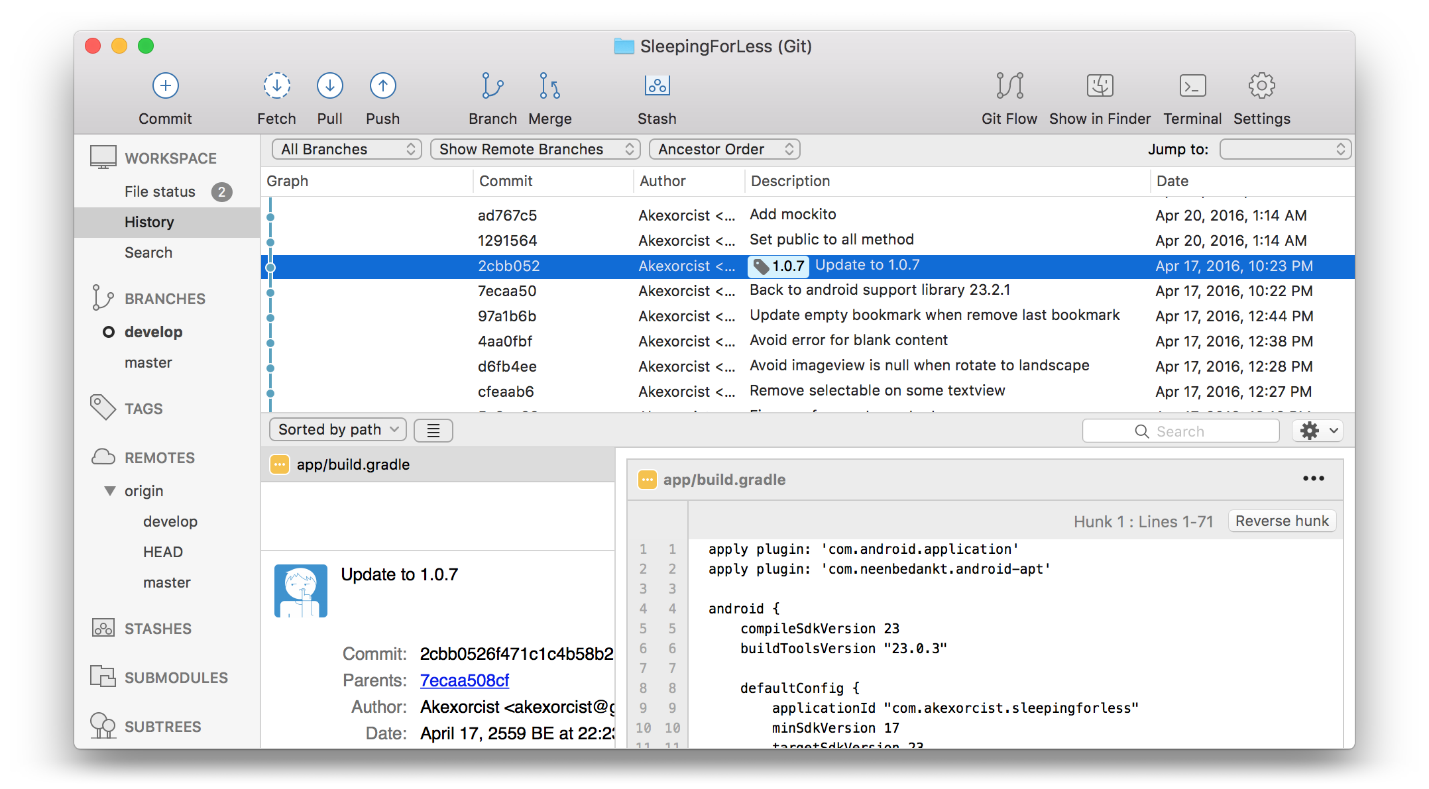
ເມື່ອ​ແກ້​ໄຂ​ສຳເລັດ​ແລ້ວ​ກໍ​ຈະ Merge ເຂົ້າ​ສູ່ master ແລະ develop ຄື​ກັບ release

ແລະ ​ນີ້​ກໍ​ຄື​ຮູບ​ແບບ​ທັງ​ໝົດ​ຂອງ Git Flow ທີ່​ຖືກ​ອອກ​ແບບ​ມາ​ໃຫ້​ຕອບ​ໂຈ​ດໃນ​ການ​ພັດທະນາ​ໂປຣ​ເຈັກ​ໃຫຍ່​ໆ ໂດຍ​ຈະ​ເຫັນ​ວ່າ Git Flow ຈະ​ຊ່ວຍ​ໃຫ້​ຂຽນ​ໂຄ້​ດ​ບໍ່​ຊັບ​ຊ້ອນ​ວຸ້ນວາຍ ບໍລິຫານ​ ແລະ​ ຈັດການ Feature ຕ່າງ​ໆ​ໄດ້​ງ່າຍ ຮອງ​ຮັບ Process ທີ່​ຕ້ອງ​ມີ​ການ​ຜ່ານ QA/Tester ກ່ອນ​ຈະ​ຂຶ້ນ​ສູ່ Production ດ້ວຍ ແລະ​ ຕອບ​ໂຈ​ດ​ສຳລັບ​ການ​ຮັບ​ມື​ກັບ Issue ຕ່າງ​ໆ​ທີ່​ສາມາດ​ເກີດ​ຂຶ້ນ​ໄດ້​ຕະຫລອດ​ເວລາ

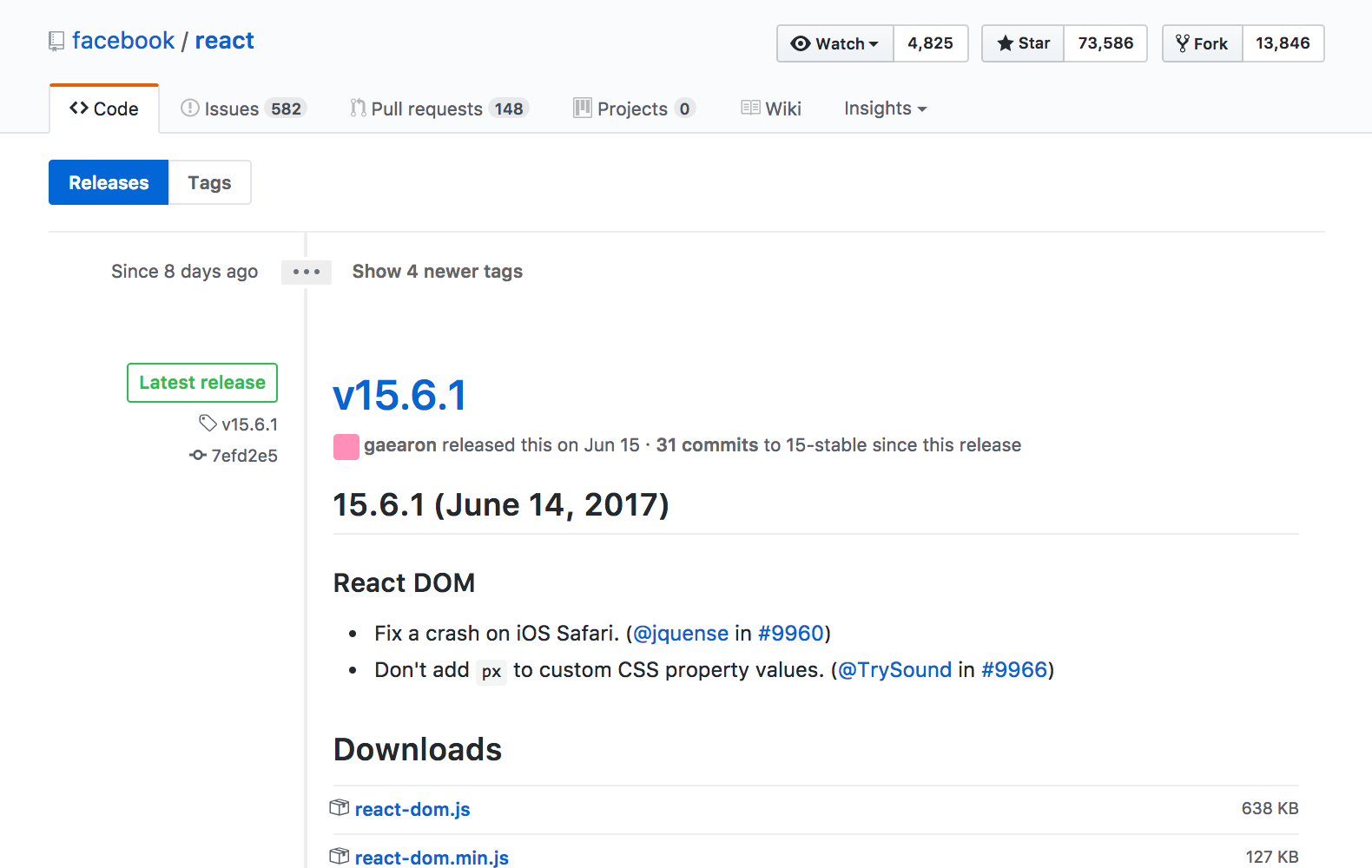
**ເພິ່ມເຕີມ**  Git Flow ອາດຈະ​ເບິ່ງ​ວຸ້ນວາຍ​ຕອນ​ທີ່​ສ້າງ Branch ຕ່າງ​ໆ​ຕາມ​ທີ່​ກຳນົດ​ໄວ້ ແຕ່​ໃນ​ຄວາມ​ເປັນ​ຈິງໃນ Git GUI ຫຼາຍ​ໆ​ຕົວ​ນັ້ນ​ມີ​ຕົວ​ຊ່ວຍ​ສ້າງ Git Flow ໃຫ້​ງ່າຍ​ຂຶ້ນອີກ​ດ້ວຍ ພຽງ​ແຕ່ກົດ​ ຕັ້ງຊື່ ແລ້ວ​ກໍ​ກົດ​ໆ

## Image for post**Tag**

ຖ້າ​ເບິ່ງ​ຈາກ​ທີ່​​ອະທິບາຍ​ໃນ Git Flow ກໍ​ຈະ​ເຫັນ​ວ່າ​ໃນ Master ຈະ​ມີ Tagຢູ່​ນຳຊຶ່ງ Tag ຈະ​ເປັນ​ຄື​ຕົວ​ຊ່ວຍ​ບອກ​ວ່າ​ອັນ​ໃດ​ຢູ່​ທີ່ Commit ໃດ ເພື່ອ​ທີ່​ວ່າ​ຈະ​ໄດ້​ຕາມ​ຫາ​ພາຍ​ຫລັງ​ໄດ້​ງ່າຍ



ໃນ​ກໍລະນີ​ທີ່​ໃຊ້ Git Flow ກໍ​ຈະ​ມີ​ການ​ໃສ່ Tag ໃຫ້​ອັດ​ໂນ​ມັດຕາມ​ຊື່ Branch ທີ່​ກຳນົດ​ໄວ້​ຕອນ Release ແລະ Hotfix ແລະ ​ຖ້າ​ໃຊ້ VCS ຂອງ​ຜູ້​ໃຫ້​ບໍລິການ​ຢ່າງ GitHub ກໍ​ຈະ​ມີ​ເມ​ນູ​ສະແດງ Source Code ຂອງ​ແຕ່​ລະ Tag ໄດ້​ທັນ​ທີ



## **Fork**

ໂດຍ​ປົກກະຕິ​ແລ້ວ ຄົົນທີ່​ສາມາດ​ເຂົ້າໄປ Commit ແລະ Push ໃນ Repository ໄດ້​ຈະ​ຕ້ອງ​ເປັນ​ຄົົນທີ່​ໄດ້ຮັບ​ອະນຸຍາດ​ເທົ່າ​ນັ້ນ ເຖິງ​ແມ່ນ​ວ່າ​ຈະ​ເປັນ Public Repository ກໍ​ຕາມ

ສົມມຸດວ່າ​ໄປ​ໃຊ້ Library ຕົວ​ໜຶ່ງ​ທີ່​ມີ Repository ຢູ່​ເທິງ GitHub ແລ້ວ Library ຕົວ​ນັ້ນ​​ມີ​ບັ​​ກ​ຢູ່ ແຕ່​ຄົນ​ເບິ່ງ​ແຍງ​ບໍ່​ວ່າງ​ມາ​ແກ້​ໄຂ​ໃຫ້​ ແລະ ຮີບ​ຮ້ອນ​ຫລາຍ​ໆ ຢາກ​ຈະ​ໃຊ້​ໄວ​ໆ​ເພື່ອ​ໃຫ້​ງານ​ສຳເລັດ ດັ່ງ​ນັ້ນ ​​ກໍ​ຕ້ອງ​ກໍ່ການ ForkRepository ຕົວ​ນັ້ນ​ມາ​ໄວ້​ທີ່ GitHub ​ແລ້ວ​ກໍ່ການ​ແກ້​ໄຂ​ໃຫ້​ຮຽບຮ້ອຍ​

ຊຶ່ງ​ການ Fork ກໍ​ຄື​ການ​ຄັດ​ລອກ Repository (ຂອງ​ຄົນ​ອື່ນ​ທີ່​ອະນຸຍາດ​ໃຫ້​ເຮົາ​ເຂົ້າໄປ​ເບິ່ງ​ໄດ້) ມາ​ເກັບ​ໄວ້​ເປັນ Repository ຂອງ​ເຮົາ ຊຶ່ງ​ທຸກ​ໆ​ຢ່າງ​ຈະ​ຄືກັນ​ທັງ​ໝົດ ຕ່າງ​ແຕ່ວ່າ​ເຮົາ​ເປັນ​ເຈົ້າ​ຂອງ Repository ຕັວ​ທີ່​ແຍກ​ອອກ​ມາ​ເທົ່າ​ນັ້ນ​ເອງ ດັ່ງ​ນັ້ນ ​ຈຶ່ງ​ສາມາດ​ແກ້​ໄຂ​ບັ​ກ​ຕັວ​ນັ້ນ​ຈາກ​ໃນ Repository ​ໄດ້​ທັນ​ທີ

## **Pull Request**

ເມື່ອ​ຜົມ​ແກ້​ບັກ​ໃນ Repository ທີ່ Fork ມາ​ຈາກ Library ຂອງ​ຄົນ​ອື່ນ​ແລ້ວ ແຕ່​ຜົມ​ກໍ​ພົບວ່າ​ຄົນ​ອື່ນ​ທີ່​ເອົາ Library ຂອງ​ຄົນ​ໆ​ນັ້ນ​ໄປ​ກໍ​ຢັງ​ເຈີ​ບັກ​​ຢູ່​ດີ ກໍ​ເລີຍ​ຢາກ​ຈະ​ຊ່ວຍ​ແກ້​ບັກ​ນີ້​ໃນ Library ດັ່ງ​ນັ້ນ​ຈຶ່ງ​ສົ່ງ Pull Requestໄປ​ໃຫ້​ຕົ້ນ​ທາງ​ເພື່ອ​ໃຫ້​ເຈົ້າ​ຂອງ Library ເຫັນ Commit ​ທີ່​ແກ້​ບັ​ກ​ຕົວ​ນັ້ນ

ເຈົ້າ​ຂອງ Library ທີ່​ເຫັນ Commit ​ກໍ​ຈະ​ມີ​ທາງ​ເລືອກ​ຢູ່​ສອງ​ທາງ

* ປ່ອຍ​ຖິ້ມ​ໄວ້
* ຮັບ Pull Request ເພື່ອ​ລວມ Commit ທີ່​ແກ້​ບັ​ກ​ແລ້ວ ເຂົ້າໄປ​ໃນ Repository

ແຕ່​ຈະ​ມີ​ເຫດຜົນ​ຫຍັງ​ທີ່​ຈະ​ບໍ່​ຮັບ Pull Request ເມື່ອ​ຮັບ Pull Request ບັກ​ໃນ Library ຂອງ​ກໍ​ຖືກ​ແກ້​ໄຂ ກໍ​ອາດຈະ​ຕ້ອງ​ໄປ Release ເປັນ​ເວີ​ຊັນ​ໃໝ່​ອີກ​ ແຕ່​ທີ່​ແນ່​ໆ​ຄື​​​ໄດ້​ກາຍ​ເປັນ​ໜຶ່ງ​ໃນ Contributor ຂອງ Library ຕົວ​ນີ້​ ແລະ​ ນີ້​ກໍ​ຄື​ໂລກ​ຂອງ Open Source ນັ້ນ​ເອງ

# ສະຫລຸບ

ບໍ່​ຮູ້​ຈະ​ສະຫລຸບ​ແບບໃດ​ດີ ເພາະການ​ຂຽນ​ຍາວຫລາຍ​ ແຕ່​ກໍ​ຕ້ອງ​ບອກ​ເລີຍ​ວ່າ Git ເຂົ້າ​ມາ​ປ່ຽນ​ຊີວິດ​ນັກ​ພັດທະນາ​ໃຫ້​ດີ​ຂຶ້ນ​ຫລາຍ ແກ້​ບັນຫາ​ເດີມ​ໆ​ໃນ Development Process ທີ່​ເຄີຍເຈັບ​ຫົວ​ມາ​ດົນ​ໃຫ້​ກາຍ​ເປັນ​ເລື່ອງ​ງ່າຍ ຈະ​ເຮັດ​ຫຍັງ​ກໍ​ເປັນ​ລະບົບ​ລະບຽບ ມີ​ບັນຫາ​ເກີດ​ຂຶ້ນ​ກໍ​ຮູ້​ໄດ້​ວ່າ​ເພາະ​ຫຍັງ​ ແລະ ​ຝີ​ມື​ໃຜ ແລະ ​ທີ່​ສຳຄັນ​ຄື​ໂລກ​ຂອງ​ນັກ​ພັດທະນາ​ສາມາດ Open Source ກັນ​ໄດ້​ທຸກ​ມື້​ນີ້ ສ່ວນຫນຶ່ງ​ກໍ​ຕ້ອງ​ຂອບໃຈ Git ​ທີ່​ເຮັດ​ໃຫ້ເປັນ​ເລື່ອງ​ງ່າຍ ເຖິງ​ແມ່ນ​ວ່າ​ຈະ​ມີ VCS ມາ​ດົນ​ແລ້ວ​ກໍ​ຈິງ ແຕ່​ດ້ວຍ​ຄວາມ​ນິຍົມ​ຂອງ Git ແລະ GitHub ຈຶ່ງ​ເຮັດໃຫ້​ເກີດ Community ຂອງ​ຊາວ Open Source ໄດ້​ຢ່າງ​ຫຼວງ​ຫຼາຍ​ນັ້ນ​ເອງ

ສຸດ​ທ້າຍ​ນີ້ ນອກ​ຈາກ​ການ​ໃຊ້ Git ເປັນ​ແລ້ວ ກໍ​ແນະ​ນຳ​ໃຫ້​ທຸກ​ໆ​ຄົນ​ລອງ​ສ້າງ​ໂປຣ​ເຈັກ ​ຫລື Library ​ແລ້ວ​ເປີດ Public ເປັນ Open Source ເທິງ GitHub ເບິ່ງ​ແລ້ວ​ຈະ​ຮູ້​ວ່າ Git ນັ້ນ​ຍັງ​ເຮັດ​ຫຍັງ​ໄດ້​ອີກ​ ຫລາຍ ເພາະ​ຍັງມີ​ອີກ​ຫຼາຍ​ໆ​ຢ່າງ​ທີ່​ຍັງ​ບໍ່​ໄດ້​ເວົ້າ​ເຖິງ ບໍ່​ວ່າ​ຈະ​ເປັນ .gitignore, Reset, Rebase, Cherry Pick, File Diff ແລະ ​ອື່ນ​ໆ​ອີກ​ຫລວງ​ຫລາຍ