Set up CppUnit Eclipse plugin (ECUT) for C++ on Ubuntu

First, install CppUnit. On Ubuntu, we can install from the repository:

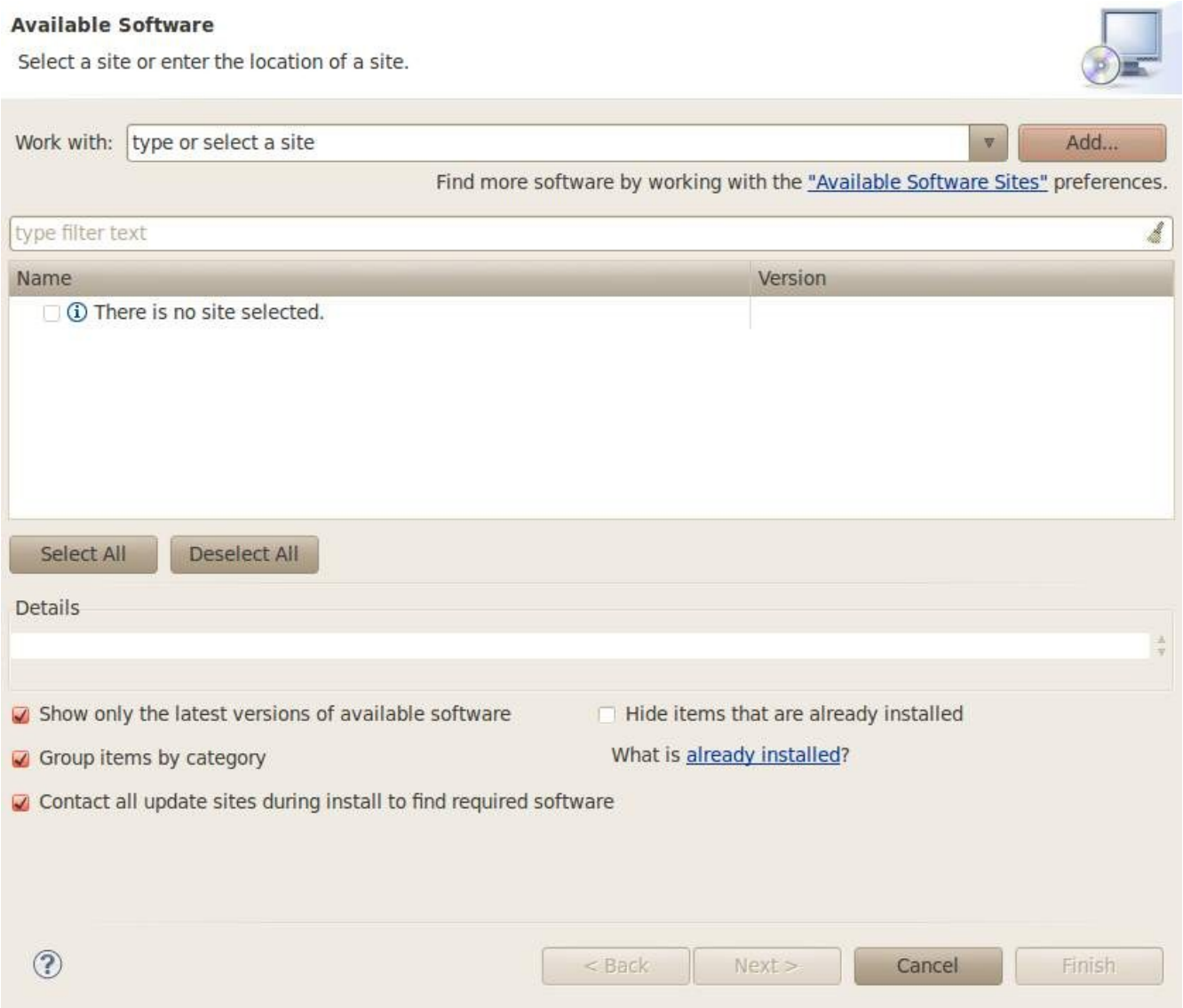
```
sudo apt-get install libcppunit-1.12-1 libcppunit-dev
```

This is for ubuntu 10.04. For other version, try

```
sudo apt-get install libcppunit*
```

Install ECUT plagin in Eclipse with the software installer in the Help menu.

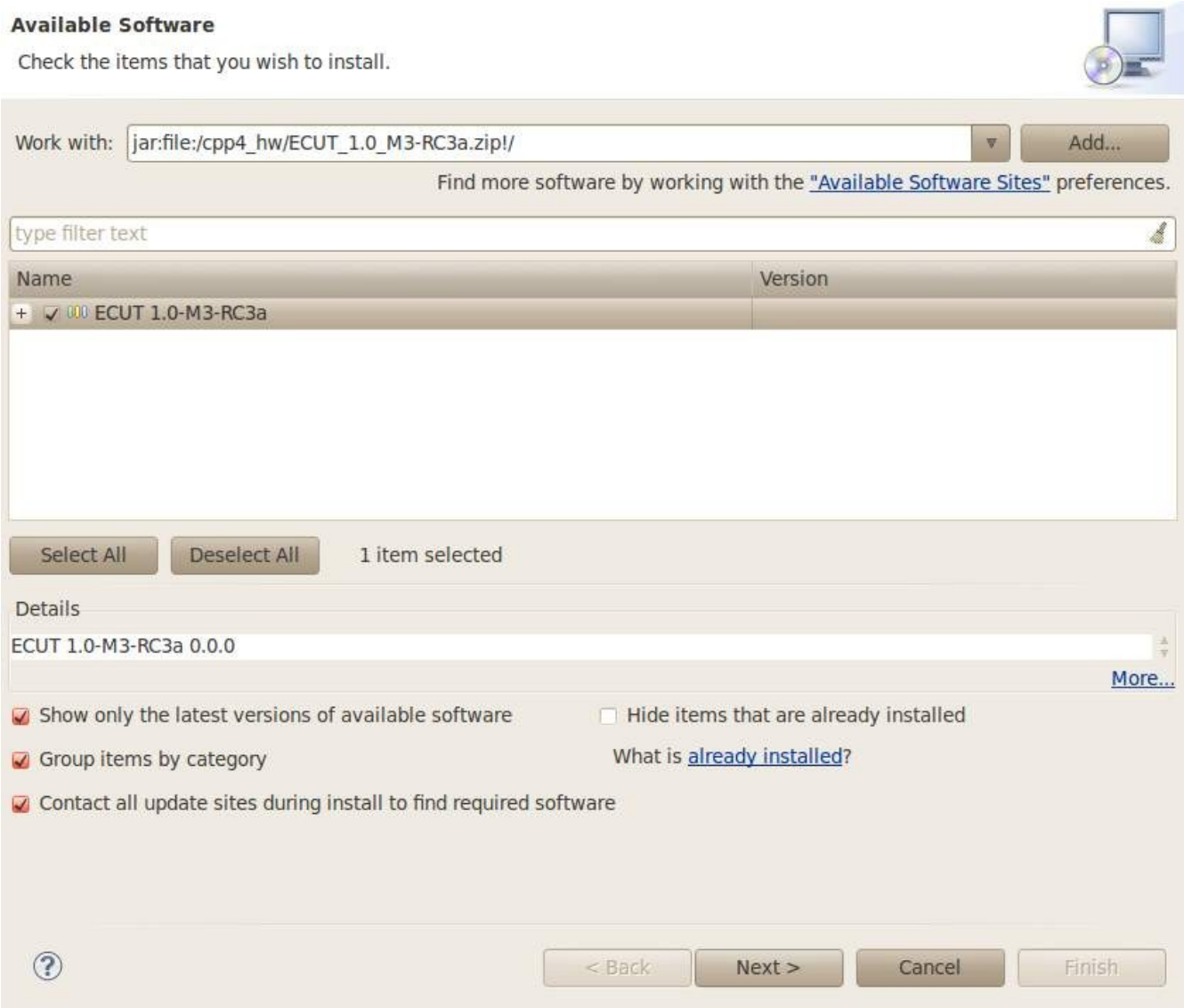
Click **Help/Install New Software...**



Click Add

Click **Archive**, find your **ECUT_1.0_M3-RC3a.zip** file, and click OK.

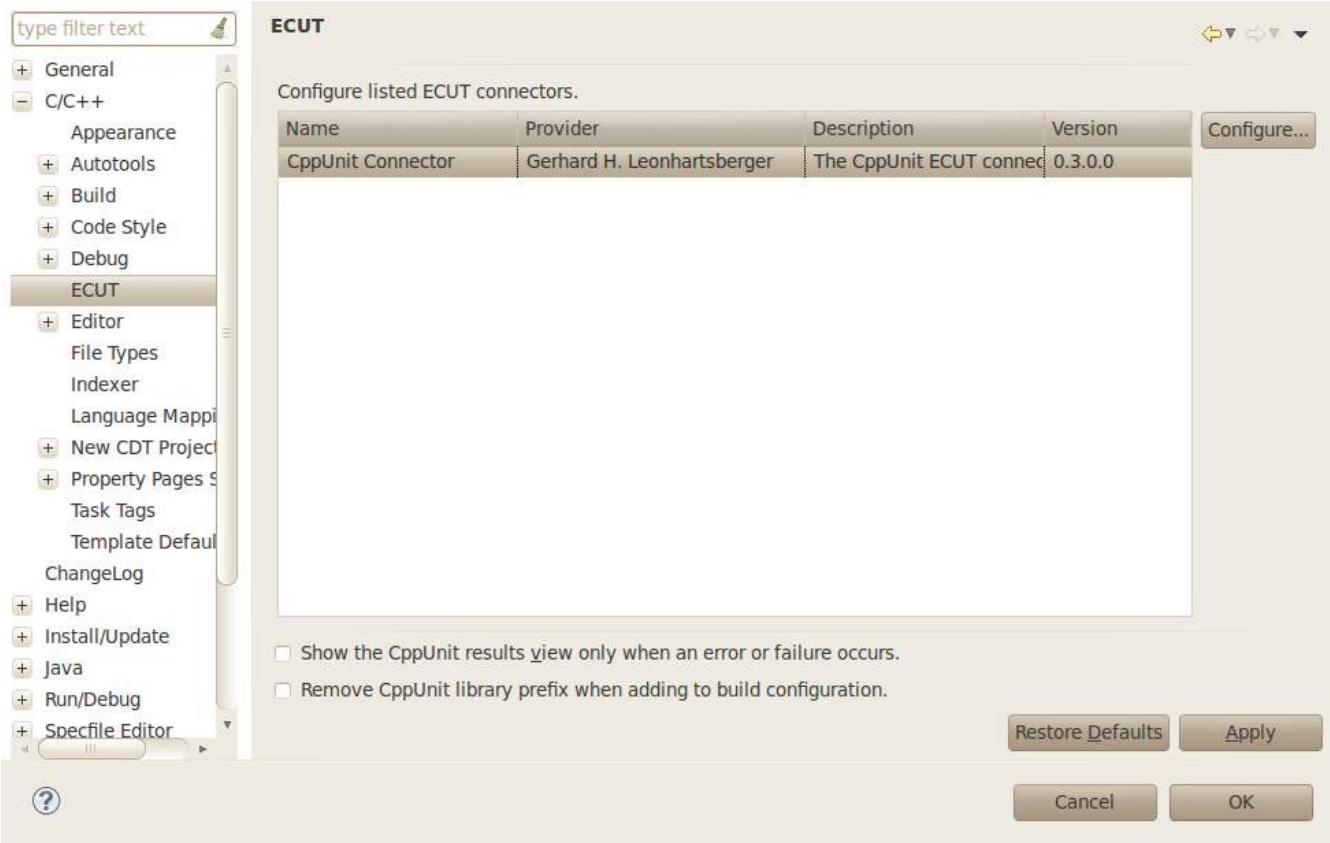
Choose ECUT 1.0-M3-Rc32, click next and fellow it to install.



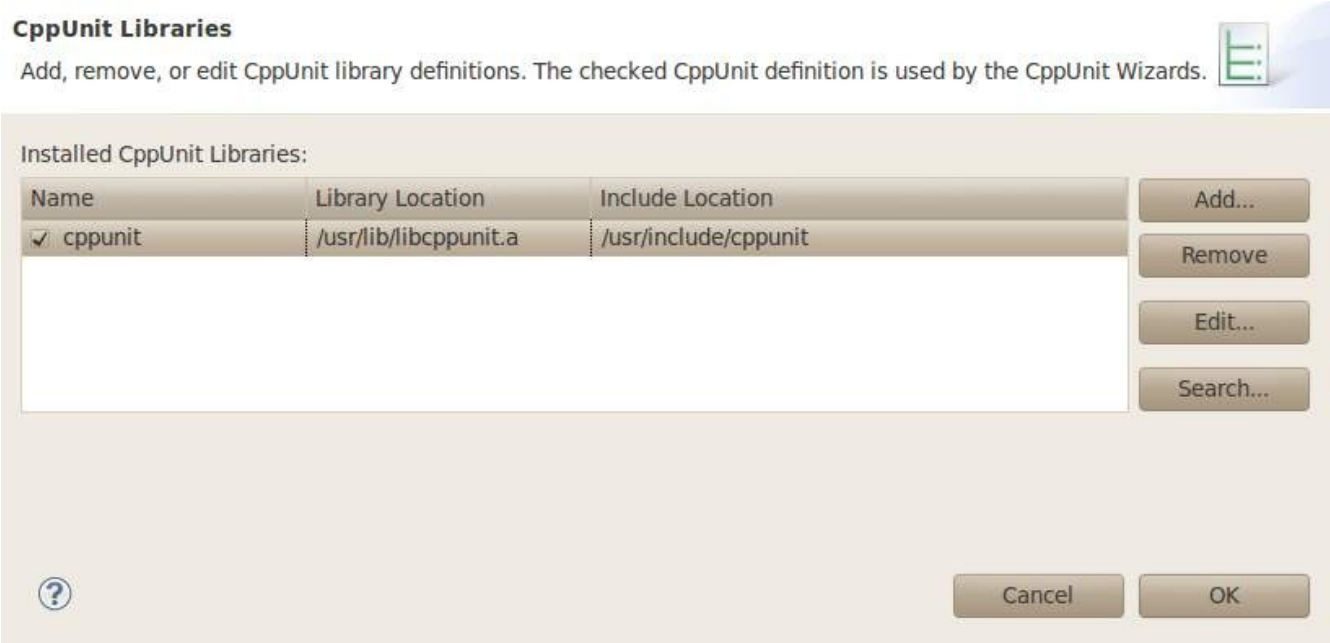
Configur the connector in Eclipse

click **Window->Preferences**

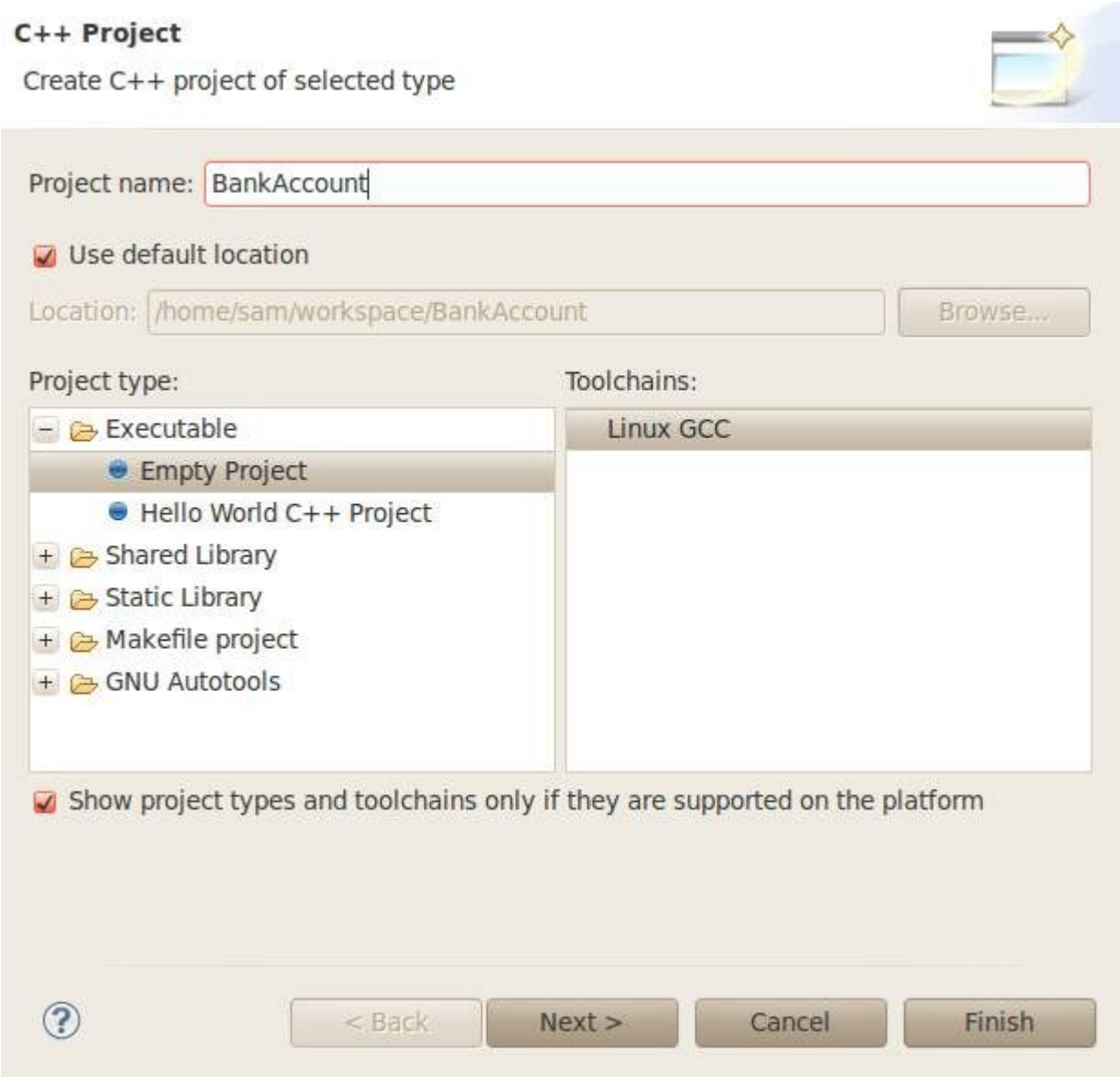
click **Configure** in **C/C++ ->ECUT**



click **Add...** to add “Library location: /usr/lib/libcppunit.a” and “Include location: /usr/include/cppunit”.



Creating a BankAccount Project



To create a project in Eclipse, select **File / New / Project...** from the main menu, expand the **C++** branch of the tree, then select **C++ Project** and click **Next**.

you should be able to expand the **Executable** branch under "Project types:" and see an entry labeled **Empty Project**. Select this option, and enter the name of the new project in the field at the top of the wizard — for this tutorial, enter `BankAccount`.

You can now click **Finish** at this point to create the project — the remaining pages of the wizard have advanced options, and we will want to keep the default settings.

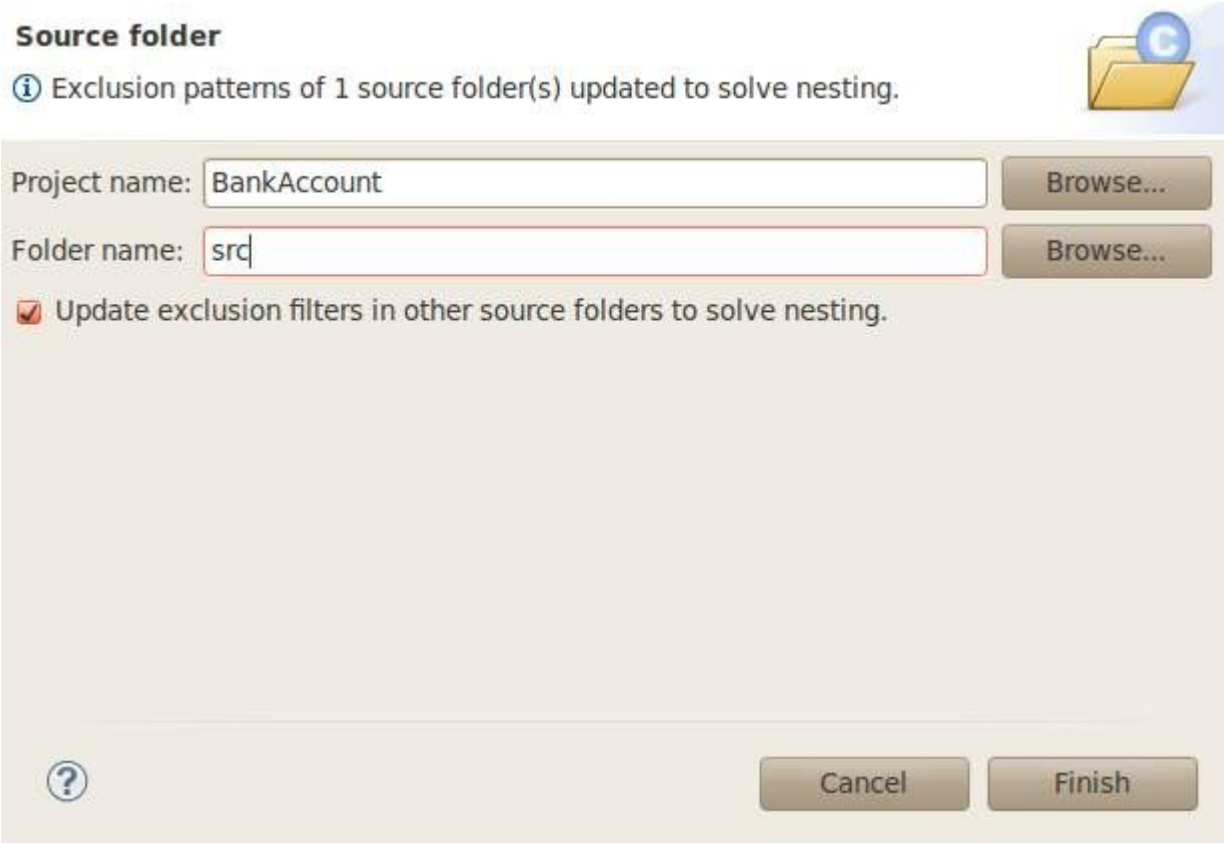
Creating the Class to Test

The new project should appear in your Eclipse workspace, in the **Project Explorer** tab. Before we can start writing test cases, clearly we need a class that we want to test. For this tutorial, consider a simple bank account class that holds a name and a dollar amount, and supports the following operations:

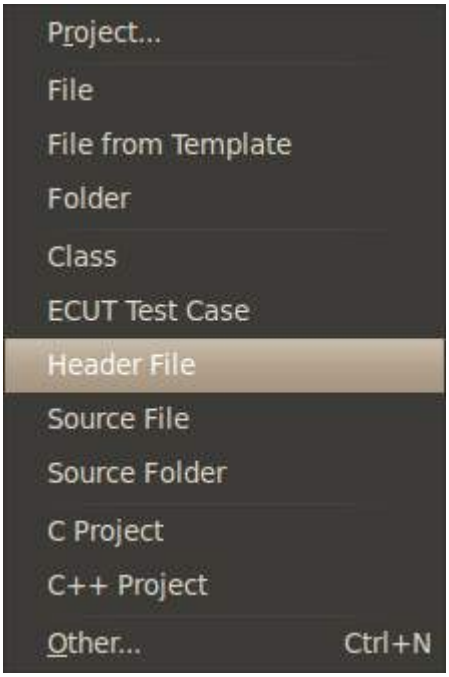
- **Construct** an empty bank account with $0
- **Construct** a bank account with a specified amount
- **Deposit** an amount into the account
- **Withdraw** an amount from the account
- Access a value containing the **current balance**
- Access a value specifying whether the account is **overdrawn** (negative balance)

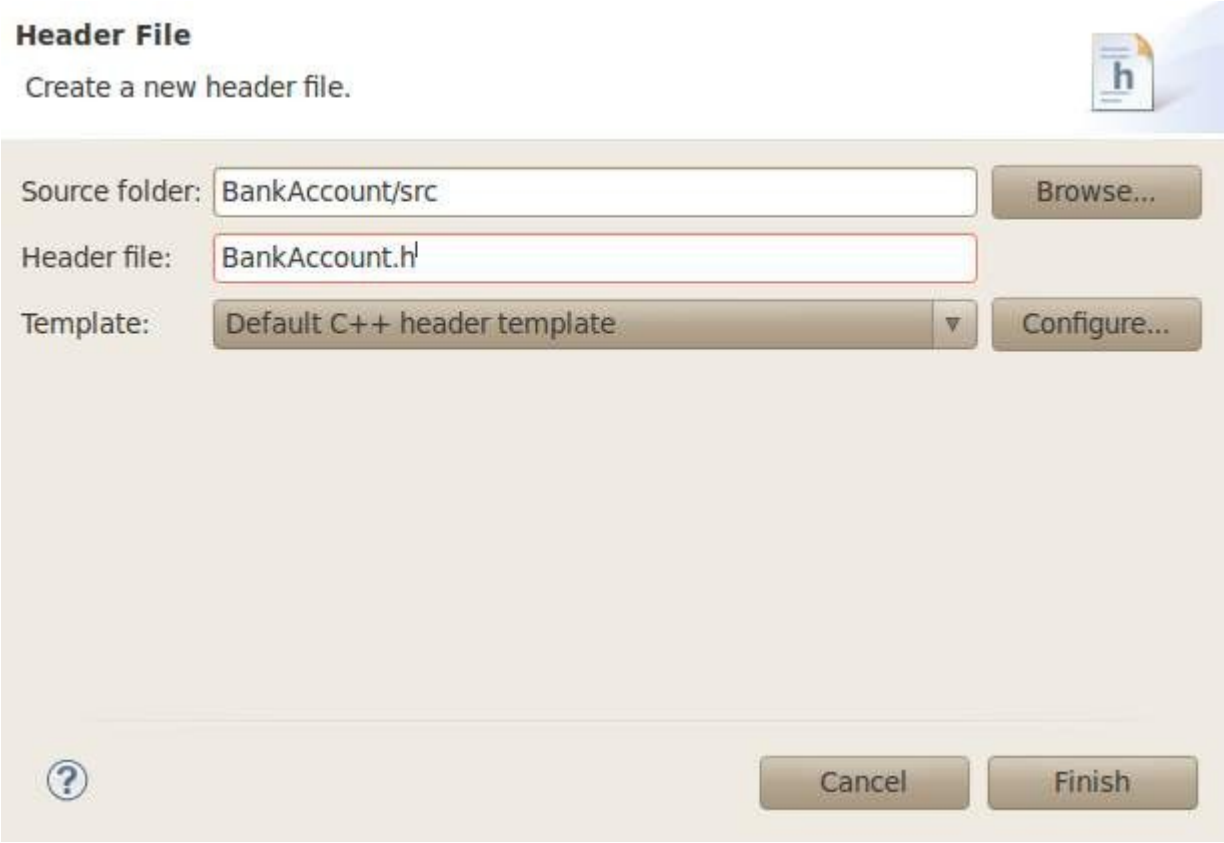
The bank account class we will be using is very limited. We store an integer dollar amount instead of dollars and cents or fractions of a dollar, because a floating-point type would have problems when checking for equality, which we need to be able to do in our test cases to verify that the balance is what we expect it to be after an operation. (Your C++ text should have more information on floating-point types and the comparison problems that are associated with them.)

Let's begin by adding the `BankAccount` class that we wish to use in our tests. Right-click on the top-level project ("BankAccount") in the Project Explorer, and choose **New/ Source Folder** from the popup menu , In the dialog, type **src** as the name of the folder and click **Finish**.



Right-click on the folder name “src” in the Project Explorer and choose **New / Header File** from the popup menu. In the dialog that appears, type `BankAccount.h` as the name of the file and click **Finish**.





This has created a new `BankAccount.h` file in the src directory of our project, and the file was automatically opened in the editor. You will see that Eclipse has already added a few lines of code to

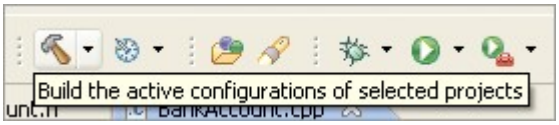
the file. These "inclusion guards" are used to prevent a header file from being included multiple times while compiling a single C++ source file. Now, add the interface for our bank account class by copying and pasting the code below into the new file.

```
// BankAccount.h

#ifndef BANKACCOUNT_H_
#define BANKACCOUNT_H_

#include <string>

class BankAccount
{
private:
	std::string accountHolder;
	int balance;

public:
	// Constructors
	BankAccount( const char* holder, int initialBalance = 0 );

	// Accessors
	int getBalance() const;
	bool isOverdrawn() const;

	// Modifiers
	void deposit( int value );
	void withdraw( int value );
};

#endif /* BANKACCOUNT_H_ */
```

Next, we need to add the implementation of these functions. Create new source file in the folder “src” of the project by right-clicking on the folder “src” and choosing **New / Source File**, and call the new file `BankAccount.cpp`. Copy the following code into this new file.

```
// BankAccount.cpp

#include "BankAccount.h"
#include <string>
using namespace std;

// Initializes a new account with the specified account holder name
// and initial balance. If the initial balance is omitted, it defaults
// to zero.
BankAccount::BankAccount( const char* holder, int initialBalance )
{
	accountHolder = holder;
	balance = initialBalance;
}

// Returns the current balance in the account.
int BankAccount::getBalance() const
{
```

```
	return balance;
}

// Returns true if the account is overdrawn (has a negative balance);
// otherwise, it returns false.
bool BankAccount::isOverdrawn() const
{
	return (balance < 0);
}

// Adds the specified amount to the balance in the account.
void BankAccount::deposit( int value )
{
	balance += value;
}

// Subtracts the specified amount from the balance in the account.
void BankAccount::withdraw( int value )
{
	balance -= value;
}
```



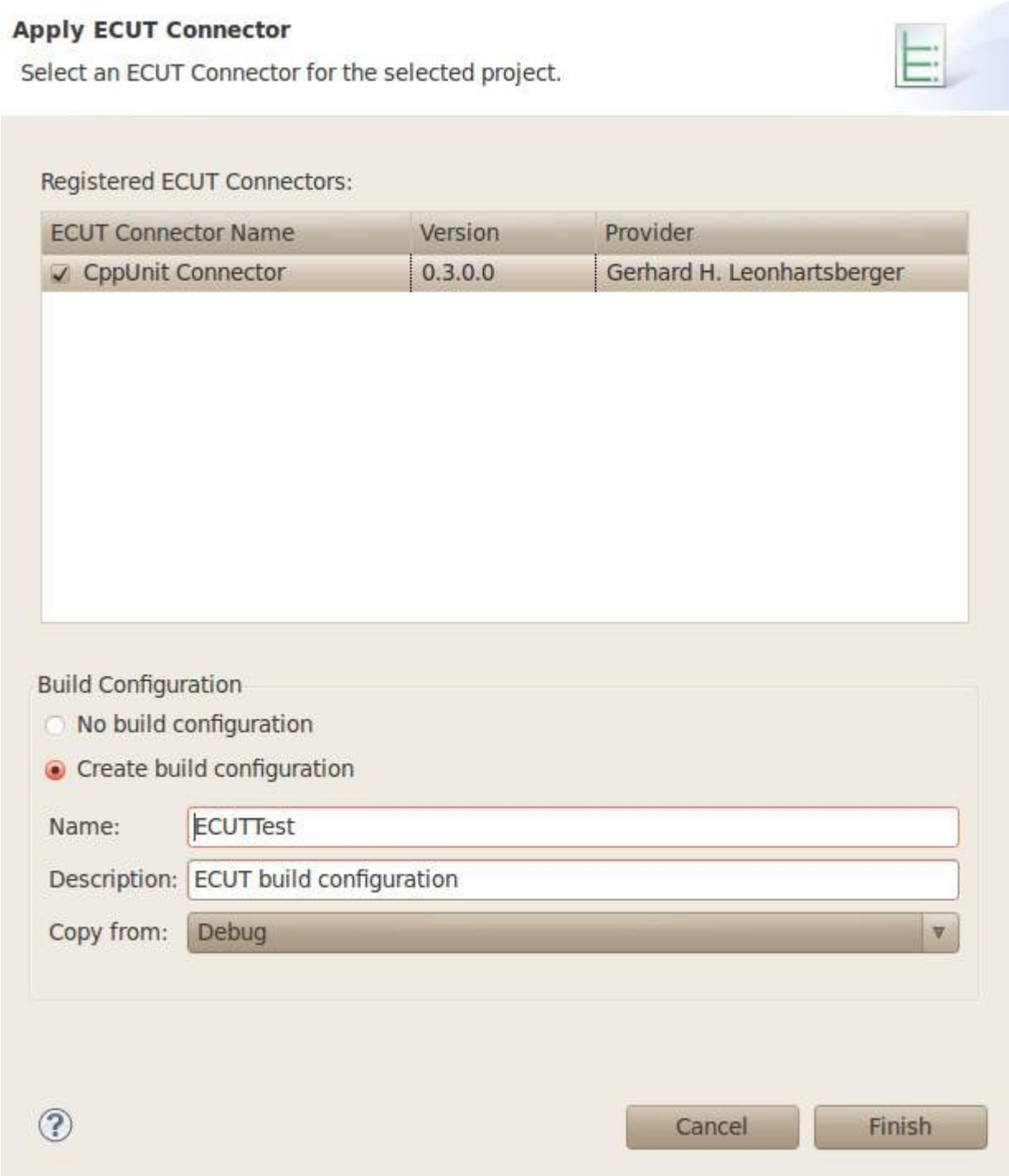
We now have a complete `BankAccount` class that we can compile and test. Unlike working on Java projects in Eclipse, you must explicitly compile your C++ projects each time you make a change to one of your source files. To do this, click on the **Build** button (shaped like a hammer, shown right) in the toolbar.

If you followed this instructions correctly, your project should build without errors. Since the test cases in a project are executed each time the project is built, We will write a test class in the next step.

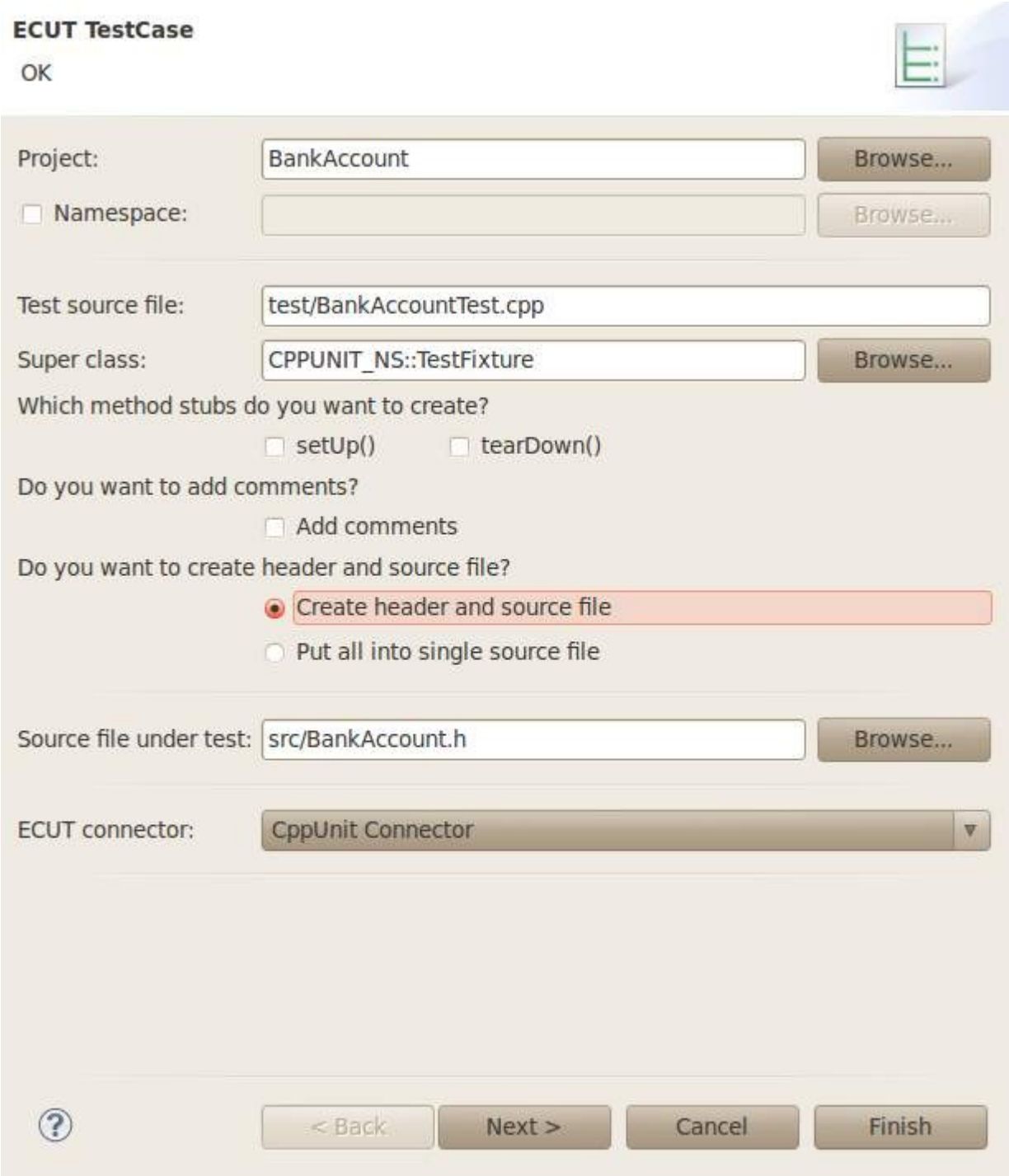
Creating the Test Cases

To create our test cases, we need to add a new `test folder` to our project that will contain test class. Right-click on the top-level project ("BankAccount") in the Project Explorer, and choose **New/ Source Folder** from the popup menu , In the dialog, type **test** as the name of the folder and click **Finish**.

Set ECUT connector to our project. Right-click on the top-level project ("BankAccount") in the Project Explorer, and choose **ECUT/Apply Connector...** from the popup menu. In the dialog, mark **CPPUNIT Connector** in Registered ECUT connectors area, select **Create build configuration** in Build Configuration and click **Finish**.



To create test class derived from CppUnit, right-click on the **BankAccout.h** in the Project Explorer, choose **New/ECUT Test Case** . In the ECUT TestCase Wizard, change **src/`BankAccountTest.cpp`** to **test/`BankAccountTest.cpp`** in Test source file area.



click **Next**,you will be presented with a tree view of all of the global functions, classes, and methods that are declared in the header file that you chose to test. This allows you to automatically generated function placeholders for all of the functions that you wish to write test cases for. Choose all of them except BankAccount constructor. click **Finish** to dismiss the wizard.

The newly generated test suite file should look like the following snippet:

File: test/BankAccountTest.h

```
#ifndef TEST_BankAccountTest
#define TEST_BankAccountTest

#ifdef ECUT_MAIN

#include "cppunit/extensions/HelperMacros.h"

class BankAccountTest : public CPPUNIT_NS::TestFixture {
public:
	void testgetBalance();
	void testisOverdrawn();
	void testdeposit();
	void testwithdraw();
	CPPUNIT_TEST_SUITE(BankAccountTest);
	CPPUNIT_TEST(testgetBalance);
	CPPUNIT_TEST(testisOverdrawn);
	CPPUNIT_TEST(testdeposit);
	CPPUNIT_TEST(testwithdraw);
	CPPUNIT_TEST_SUITE_END();
};

#endif /* ECUT_MAIN */

#endif /*BANKACCOUNTTEST*/
```

File: test/BankAccountTest.cpp

```
#ifdef ECUT_MAIN
#include "BankAccountTest.h"

#include "BankAccount.h"

CPPUNIT_TEST_SUITE_REGISTRATION(BankAccountTest);


void BankAccountTest::testgetBalance() {
	CPPUNIT_ASSERT(false);
}

void BankAccountTest::testisOverdrawn() {
	CPPUNIT_ASSERT(false);
}

void BankAccountTest::testdeposit() {
	CPPUNIT_ASSERT(false);
}

void BankAccountTest::testwithdraw() {
	CPPUNIT_ASSERT(false);
}


#endif /* ECUT_MAIN */
```

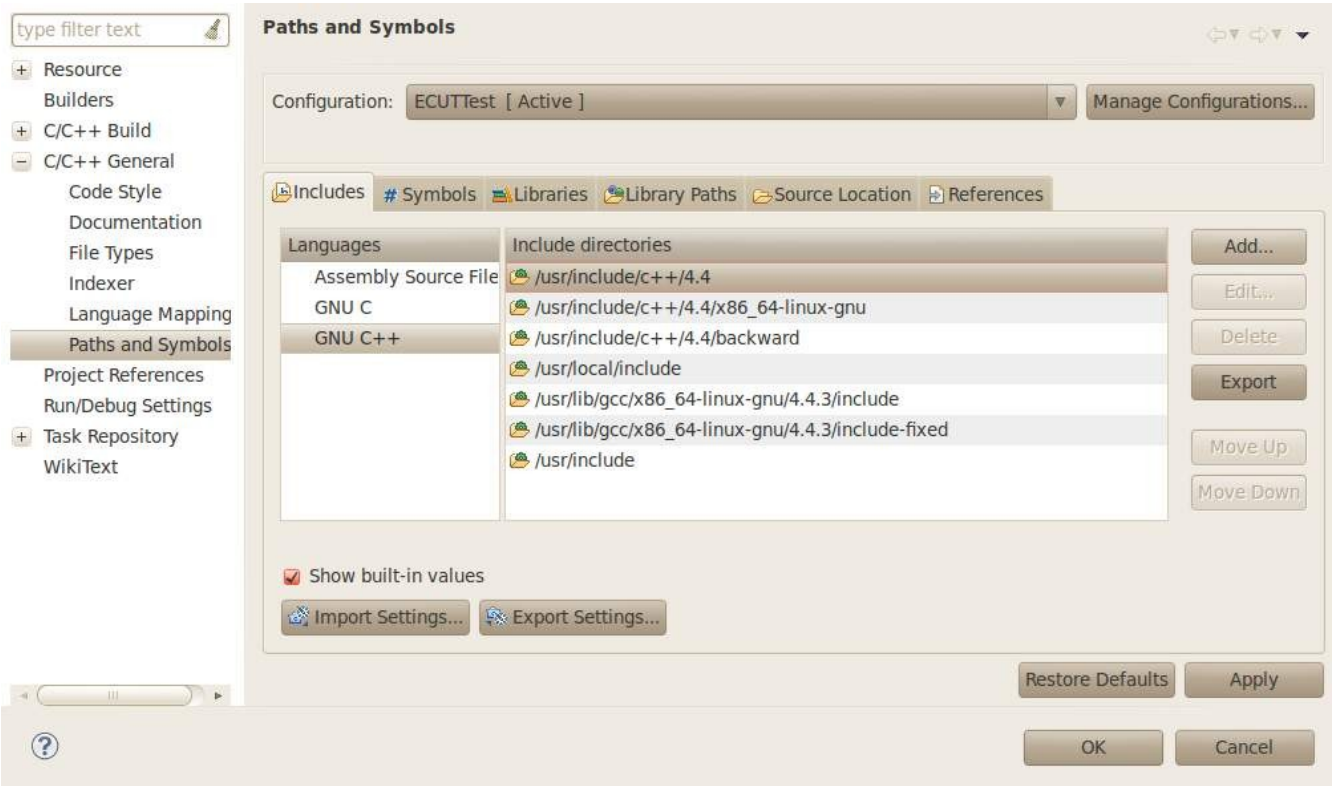
There are several points of interest here. We include the `cppunit/extensions/HelperMacros.h` header to make available the definition of the `CPPUNIT_NS::TestFixture` class, from which all test

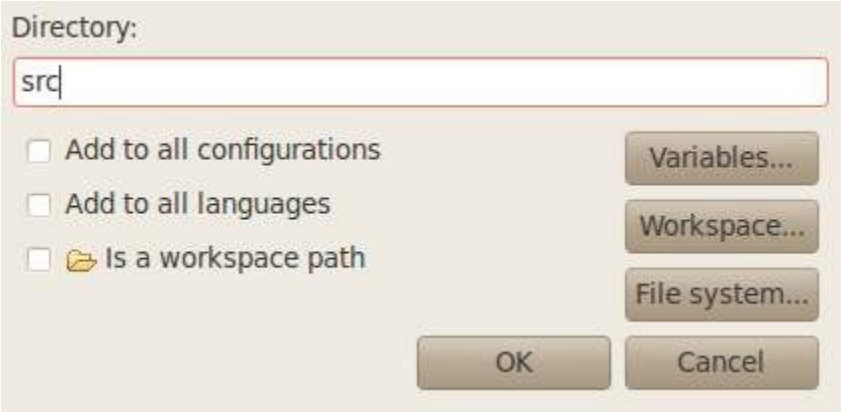
case classes must inherit. Additionally, we must obviously include the header for the `BankAccount` class that we are testing.

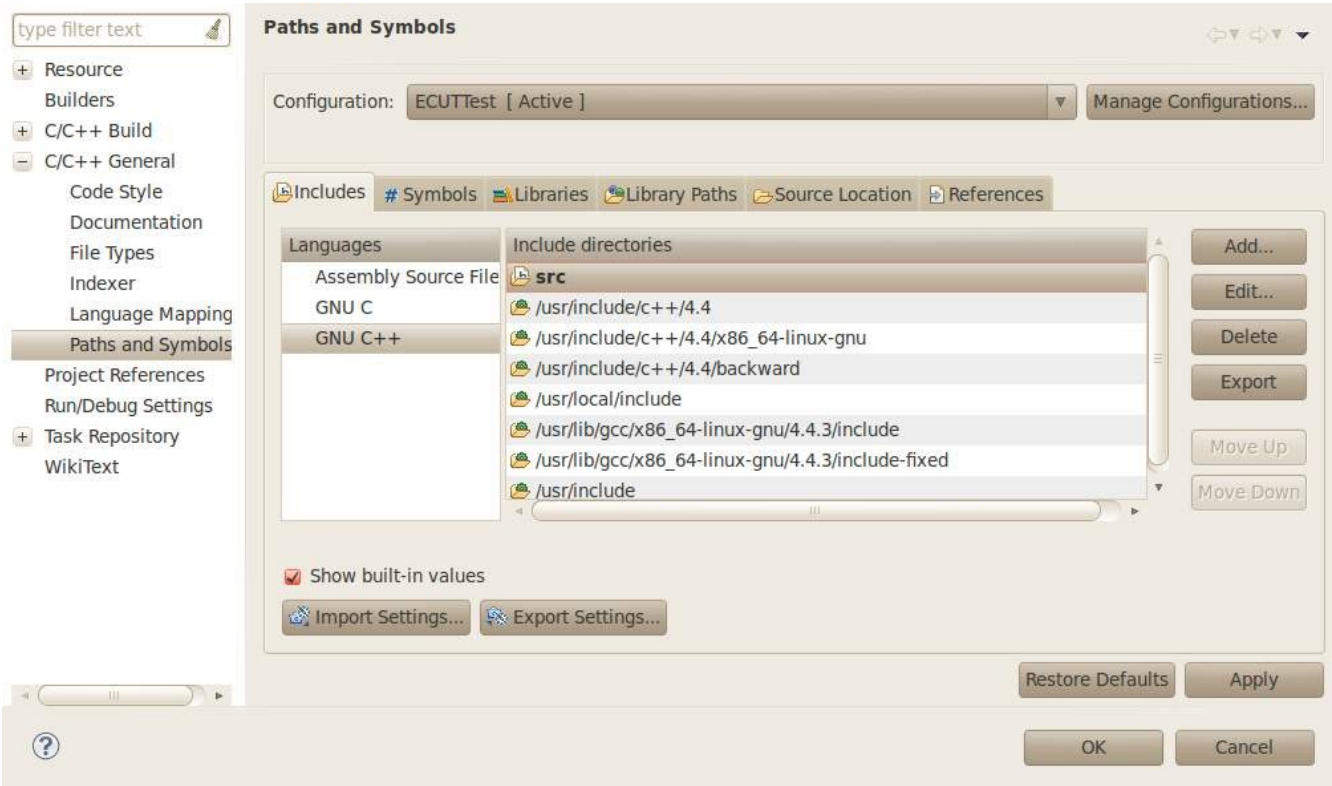
To build correct. We need set header path in ECUTTest configuration.

Right click on the top-level project ("BankAccount") in the Project Explorer, and choose **Properties.**

In **C/C++ general / Path and Symbols.** Click **GNU C++** under **languages,** and click **Add** to add **src** in it. And click **OK** to finish.



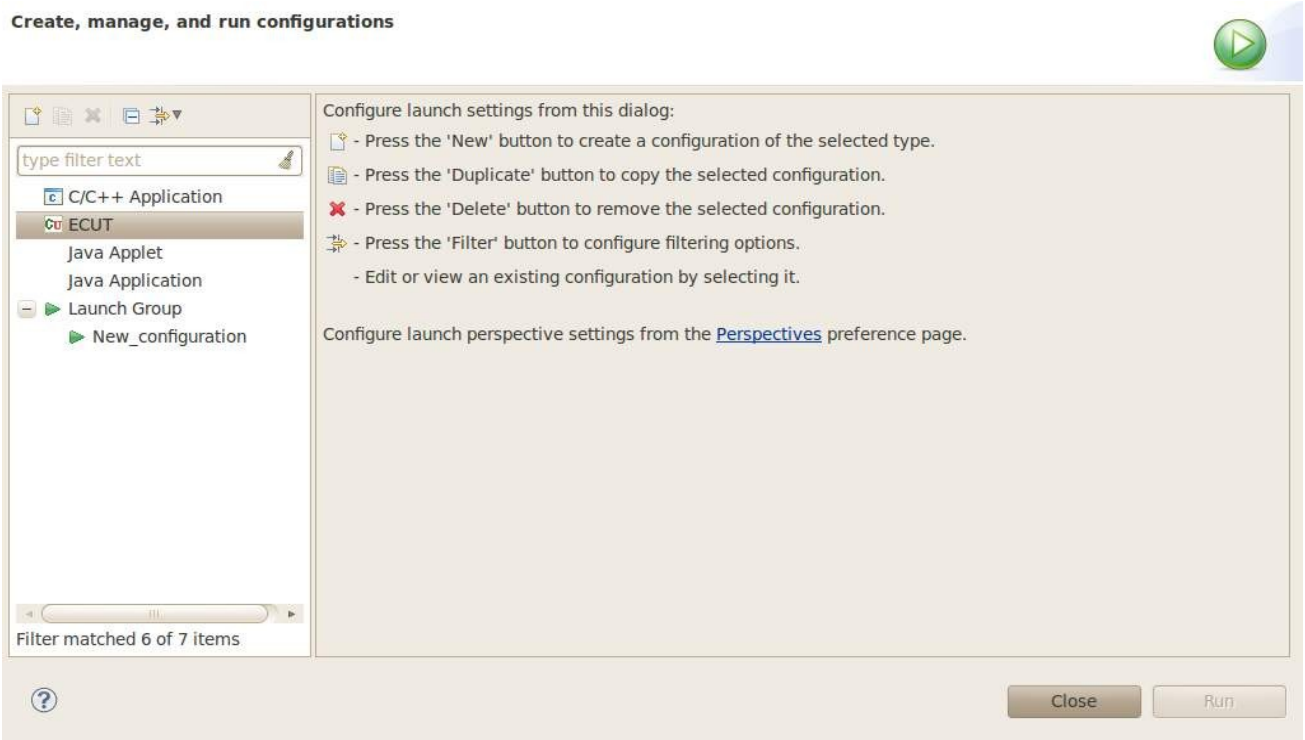




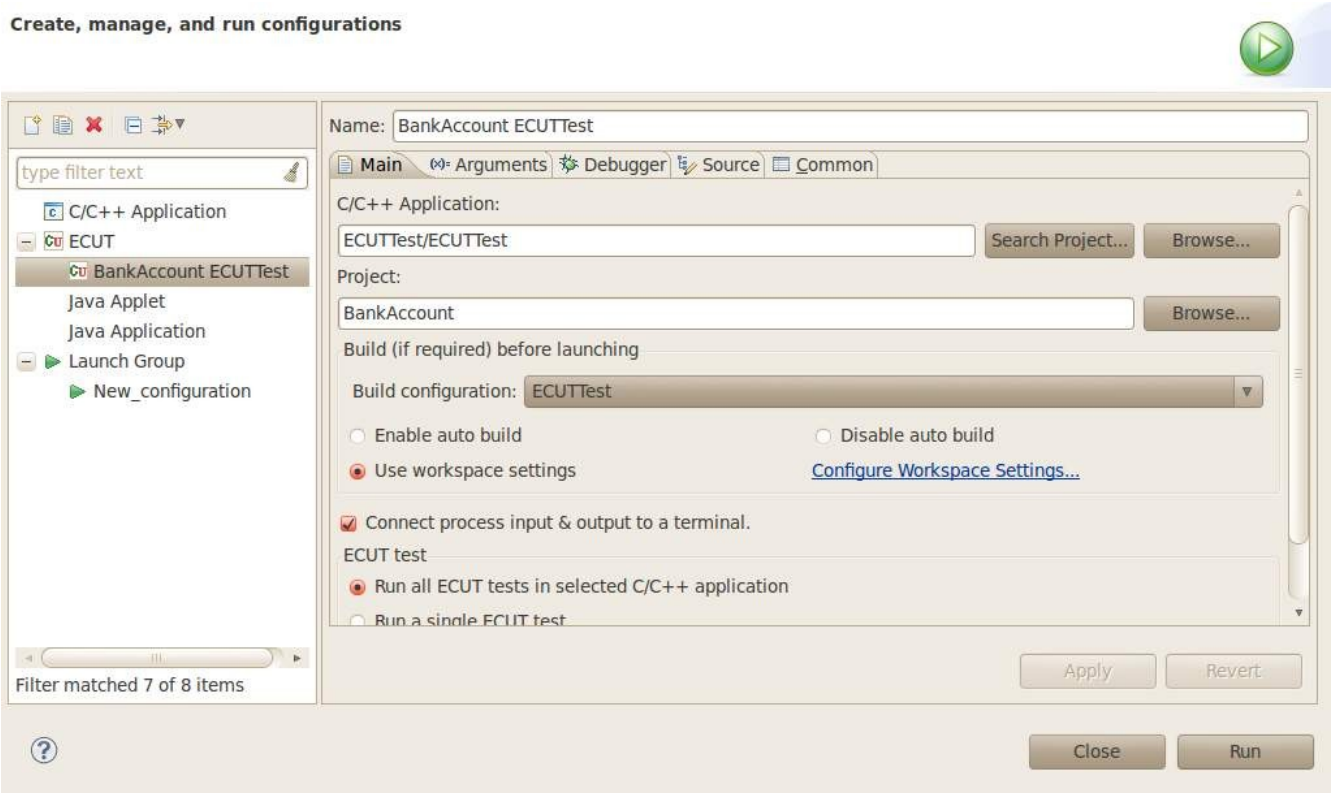
Click build icon  to build project.

To run the test, we need add a new Run configuration.

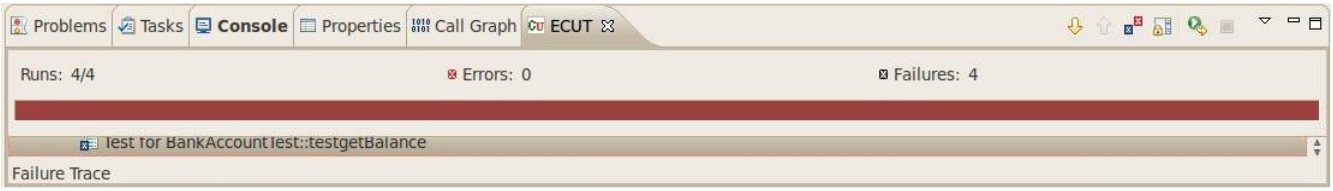
Click **Run/Run Configurations... ,**



Right click **ECUT,** and click **New** to create a new configuration see below.



Click **Run** the see the test result. This time all of them should be false. We'll change test codes at next step.



Now we will modify the test code to do the real test.

Change the four functions below. The better way is modifying one function at begin, rebuilding and running the test. This time there are three failures. And then try others.

```
void BankAccountTest::testgetBalance() {

      // An account initialized with a name only should start
      // empty.
      BankAccount emptyAccount( "Empty Account" );
      CPPUNIT_ASSERT( emptyAccount.getBalance() == 0 );
      // An account initialized with a balance should obviously
      // have that balance.
      BankAccount nonemptyAccount( "Nonempty Account", 12345 );
      CPPUNIT_ASSERT( nonemptyAccount.getBalance() == 12345 );

}
```

```
void BankAccountTest::testdeposit() {

      // Start Joe out poor, with $0.00.
      BankAccount account( "Joe Hokie" );
      // Deposit $5.00 in his account.
      account.deposit( 5 );
      // Hopefully the bank is keeping track of his money properly.
      CPPUNIT_ASSERT( account.getBalance() == 5 );
      // Have Joe deposit a little more, and verify.
      account.deposit( 100 );
      CPPUNIT_ASSERT( account.getBalance() == 105 );

}

void BankAccountTest::testwithdraw() {

      // Start Francois out moderately wealthy.
      BankAccount account( "Francois Hokie", 5000 );
      // Francois is a big spender, who makes several withdrawals.
      account.withdraw( 100 );
      account.withdraw( 200 );
      account.withdraw( 300 );
      account.withdraw( 400 );
      account.withdraw( 500 );
      // He should have $3500 left now.
      CPPUNIT_ASSERT(account.getBalance() == 3500);

}

void BankAccountTest::testisOverdrawn() {

      // Hiroyuki isn't very responsible with his money, so
      // he is going to withdraw more than he has available.
      BankAccount account( "Hiroyuki Hokie", 115 );
      account.withdraw( 200 );

      CPPUNIT_ASSERT( account.getBalance() < 0 );
      CPPUNIT_ASSERT( account.getBalance() == -85 );
      CPPUNIT_ASSERT( account.isOverdrawn() );

      // Later, a Wall Street financial expert takes Hiroyuki
      // under his wing and teaches him how to properly manage
      // his funds.  He makes a large deposit to get his life
      // back on track.
      account.deposit( 1000 );

      CPPUNIT_ASSERT( account.getBalance() > 0 );
      CPPUNIT_ASSERT( account.getBalance() == 915 );
      CPPUNIT_ASSERT( !account.isOverdrawn() );

}
```

After that, rebuild, click  ,and select **BankAccount ECUTTest** to run.

The result is:

Add a new Test case

Before we begin to add test cases to this class, there are a couple coding conventions that you must follow when writing test cases. A method that is intended to be executed as a test case must have a name that begins with `test`, a return type of `void`, and takes no arguments.

Now we will add a method to the class that perform various operations and test.

First, add a new test method in header file.

Edit BankAccountTest.h, add testVariousActivity() declaration and use CPPUNIT_TEST(testVariousActivity) macro to define it as a test between CPPUNIT_TEST_SUITE() and CPPUNIT_TEST_SUITE_END();

```
class BankAccountTest : public CPPUNIT_NS::TestFixture {
public:
      void testgetBalance();
      void testisOverdrawn();
      void testdeposit();
      void testwithdraw();
      void testVariousActivity();

      CPPUNIT_TEST_SUITE(BankAccountTest);
      CPPUNIT_TEST(testgetBalance);
      CPPUNIT_TEST(testisOverdrawn);
      CPPUNIT_TEST(testdeposit);
      CPPUNIT_TEST(testwithdraw);
      CPPUNIT_TEST(testVariousActivity);
      CPPUNIT_TEST_SUITE_END();
};


Edit BankAccoutTest.cpp, add code below

void BankAccountTest::testVariousActivity() {

      // Start Sabrina out with $75, and then monitor her
      // account over some deposits and withdrawals.
      BankAccount account( "Sabrina Hokie", 75 );
      account.deposit( 50 );
      CPPUNIT_ASSERT( account.getBalance() == 125 );
      account.withdraw( 10 );
      CPPUNIT_ASSERT( account.getBalance() == 115 );
      account.withdraw( 42 );
      CPPUNIT_ASSERT( account.getBalance() == 73 );
      account.deposit( 534 );
      CPPUNIT_ASSERT( account.getBalance() == 607 );

}
```

Rebuild and run the test.