



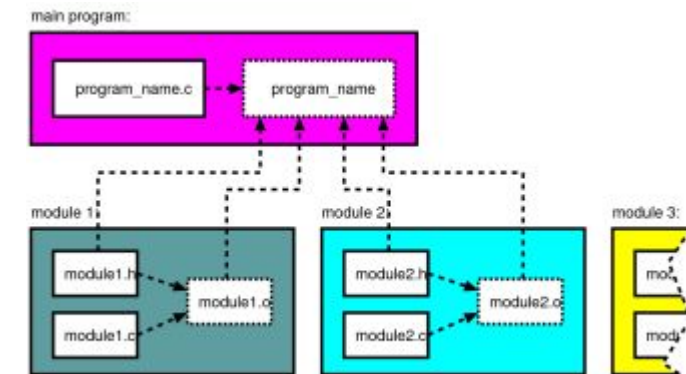
**Yrkes
Akademin**
Vi hjälper dig att lyckas!

Modularity and Software Testing

Programming and Development of Embedded Systems

Modularity

- ❖ A module is a self-contained part of a system that has
 - A well-defined interface and
 - An implementation which hides the code and any other private details to provide some functionalities for the system
- ❖ Modularity helps to
 - Organize large programs in smaller parts (modules)
 - Make testable, understandable and reusable code
 - Provide clear and flexible interfaces
 - Have separate compilation and make changes easier
 - Hide implementation and support operations on data structure
 - Encapsulate abstract data type and hide private data
 - An abstract data type is defined indirectly, only by the operations that may be done on it (e.g. **FILE**)



Modularity

❖ Types of modules

➤ Single-instance module

- A module that has a single set of data to manage
- To encapsulate the module's private data, use **static** file scope variables

➤ Multiple-instance module

- A module that has to manage different sets of data for different clients
- To encapsulate the module's **abstract data type** and private data
 - Using **typedef** of a forward declared **struct** in the header (.h) file like

`typedef struct module_Type module_Type;`

- Inside the source (.c) file, define the struct elements for **module_Type**
- A **Pointer** to the incomplete type can be passed around.

```
module1_Type *elems[100]; /* ok */
```

- ◆ No code can dereference the pointer

```
module1_Type elems[100]; /* ERR */
```

Abstract data type	
module.h	module.c
<code>typedef struct module_Type module_Type;</code>	<code>struct module_Type { int field1; int field2; ... };</code>

Modularity and Software Testing

- ❖ To make a function in a module private, use **static** file scope in the .c file
- ❖ If it is required, make the private data accessible through the module's public interface
 - By defining a set of function prototypes in the header (.h) file
- ❖ To make a module private in another module, include it in the source file, otherwise in the header file
- ❖ When doing TDD to create modular C, we will use these files and conventions
 - The **header file** which defines the module's interface
 - For single-instance modules, the header file is made up of function prototypes
 - For abstract data types, in addition to function prototypes, a typedef is created for a pointer to a forward declared struct.
 - Hiding the struct hides the data details of the module
 - The **source file** contains the implementation of the interface
 - It includes any needed private helper functions and hidden data
 - For ADTs, the forward declared struct members are defined in the source file

Modularity and Software Testing

- ❖ When doing TDD to create modular C ...
 - The **test file** holds the test cases
 - Keep test code and production code separate
 - Each module has at least one test file
 - Module **initialization** and **cleanup** functions
 - Every module that manages hidden data should have initialization and cleanup functions
 - Like **Create** and **Destroy** functions for each module
 - ADTs certainly need them with their totally hidden internals
 - ◆ Memory for the data is allocated dynamically
 - Modules that is made up of stand-alone functions, like **strlen** and **sprintf**, that keep no internal state won't need initialization and cleanup

Modularity

❖ Module Example

```
#ifndef MODULE_H
#define MODULE_H

#ifdef __cplusplus
extern "C"
{
    /* ===== */
    /* ===== include files ===== */
    /* ===== */

    /* Inclusion of system and local header files goes here */

    /* ===== */
    /* ===== constants ===== */
    /* ===== */

    /* #define and enum statements go here */

    /* ===== */
    /* ===== public data ===== */
    /* ===== */

    /* Definition of public (external) data types go here */

    /* ===== */
    /* ===== public functions ===== */
    /* ===== */

    /* Function prototypes for public (external) functions go here */

#ifdef __cplusplus
}
#endif
#endif /* MODULE_H */
```

```
/* ===== */
/* ===== include files ===== */
/* ===== */

/* Inclusion of system and local header files goes here */

/* ===== */
/* ===== constants ===== */
/* ===== */

/* #define and enum statements go here */

/* ===== */
/* ===== global variables ===== */
/* ===== */

/* Global variables definitions go here */

/* ===== */
/* ===== private data ===== */
/* ===== */

/* Definition of private datatypes go here */

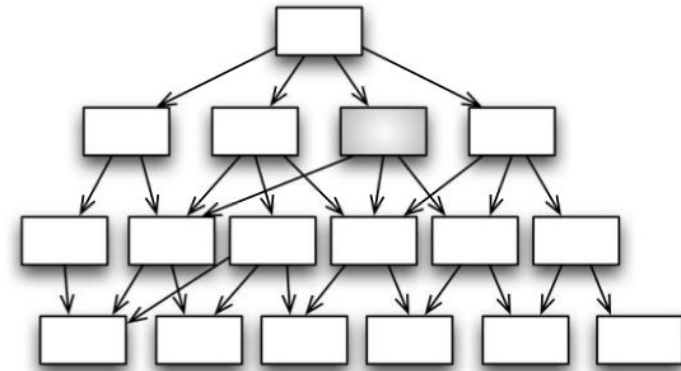
/* ===== */
/* ===== private functions ===== */
/* ===== */

/* Function prototypes for private (static) functions go here */

/* ===== */
/* ===== All functions by section ===== */
/* ===== */

/* Functions definitions go here, organised into sections */
```

- **Single Responsibility Principle**
- **Open Closed Principle**
- **Liskov Substitution Principle**
- **Interface Segregation Principle**
- **Dependency Inversion Principle**



- The module should have a single responsibility
- It should do one thing and do it well
- It should have a single reason to change
- It leads to modules with good cohesion and understandability

Modules & SOLID Design

❖ Open Closed Principle (OCP)

- The module should be open for extension but closed for modification
- It can be extended by adding new code, rather than modifying existing code
- Like a USB port can be extended to be able to plug any compliant USB devices into the port (by providing different drivers) but does not need to be modified to accept a new device

❖ Liskov Substitution Principle (LSP)

- A client modules should not care which kind of server module it is working with
- Modules with the same interface should be substitutable without any special knowledge in the calling code
- E.g. the **client** code should not behave differently when interacting with the **server's test double**

Modules & SOLID Design

❖ Interface Segregation Principle (ISP)

- The client modules should not depend on fat interfaces
- Interfaces should be tailored and very focused to the client's needs
- It limits dependencies, makes code more easily tested and ported

❖ Dependency Inversion Principle (DIP)

- High-level modules shouldn't depend on low-level modules
- Dependencies should be broken with abstractions and interfaces
 - We hide the implementation details behind an interface
 - A client calls a server through a function pointer
 - A server calls a client back through a function pointer
 - An ADT hides the details of a data type

Modules & SOLID Design

❖ SOLID C Design Models

➤ Single-instance module

- Encapsulates a module's internal state when only one instance of the module is needed

➤ Multiple-instance module

- Encapsulates a module's internal state and create multiple instances of the module's data

➤ Dynamic interface

- Allows a module's interface functions to be assigned at runtime
- Overriding the functions using function pointers

➤ Per-type dynamic interface

- Allows multiple types of modules with the same interface to have unique interface functions
- E.g. By providing different drivers for a USB port

Exercise 11

- Using TDD develop a multiple instance unique and sorted linked list module.

Extra exercise

- Using TDD develop a multiple instance circular buffer module. Size of each buffer is specified during creation of the buffer.

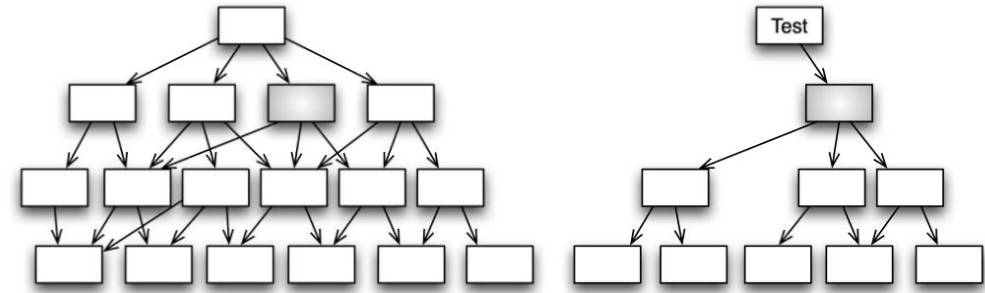
Modules & Test Double

- ❖ **CUT** : Code Under Test
- ❖ A **dependency** is a function, data, module, library or device outside the CUT
 - Real code has dependencies. A module interacts with several others to do its job.
 - Dependencies make test automation difficult and expensive
- ❖ A **collaborator** is a dependency that the CUT depends upon
- ❖ A **test double** impersonates some dependencies during the test of the CUT
 - The CUT does not know it is using a test double
 - The CUT interacts with the double in the same way it interacts with the real collaborator
 - The test double is a stub taking the place of the actual production code
 - In TDD, the test doubles are used to facilitate automated testing

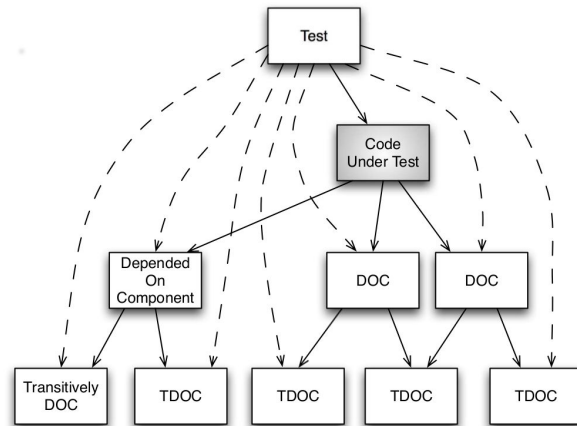
Modules & Test Double

❖ Breaking Dependencies

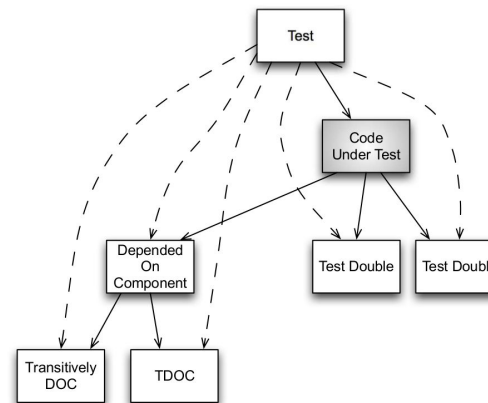
- Using of interfaces
- Using of hierarchical design
- Encapsulation and hiding data
- Less reliance on unprotected global data



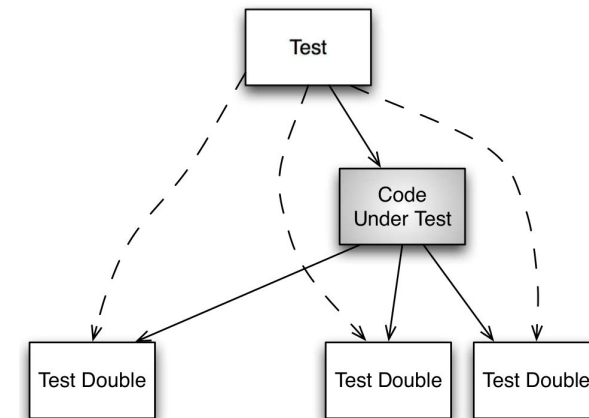
Dependencies of the gray module



Test Dependency Mess



Using test doubles and real collaborators



Manage test dependencies with test doubles

Modules & Test Double

- ❖ The rule of thumb
 - Use the real code when you can and use a test double when you must
- ❖ When to Use a Test Double
 - Hardware independence
 - Testing independently from the hardware
 - Providing a wide variety of inputs into the core of the system that may be difficult to do it in the lab
 - Inject difficult to produce input(s)
 - Some computed or hardware-generated event scenarios may be difficult to produce
 - Speed up a slow collaborator
 - A slow collaborator, such as a database or a network service

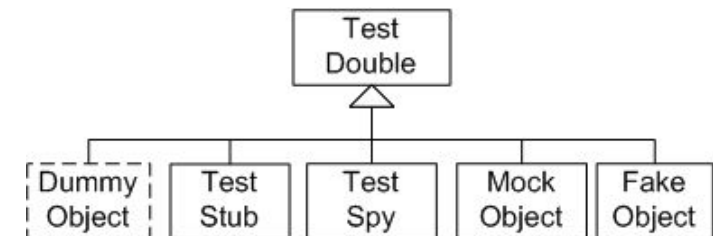
Modules & Test Double

❖ When to Use a Test Double ...

- Dependency on something under development
- Dependency on something volatile
 - Like an alarm clock.
 - You have an event that is supposed to get triggered at 03:42
- Dependency on something that is difficult to configure
 - A database is an example of a DOC that you could test with but is difficult to set up

❖ Test Doubles Variations

- There are different types of doubles like
 - Test Stub, Test Spy, Fake Object, Mock Object and etc.



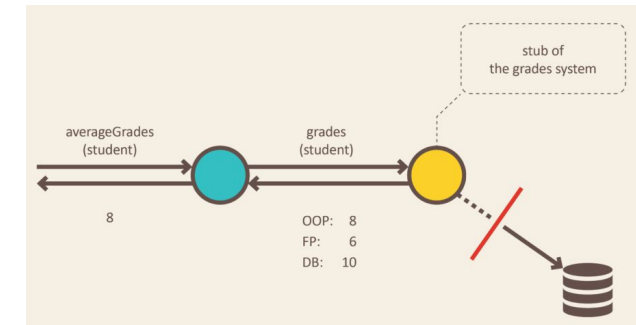
Modules & Test Double Variations

❖ Test dummy

- It is provided to satisfy the compiler, linker or runtime dependency

❖ Test stub

- It returns a value, as directed by the current test case
- It holds predefined data and uses it to answer calls during tests
- It is used when we cannot or don't want to involve objects that would answer with real data



❖ Test spy

- It captures the parameters passed from the CUT so the test can verify that the correct parameters have been passed to the CUT
- It can also feed return values to the CUT just like a test stub

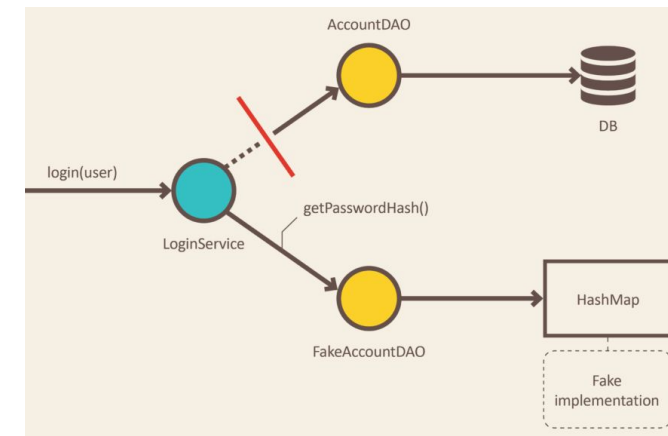
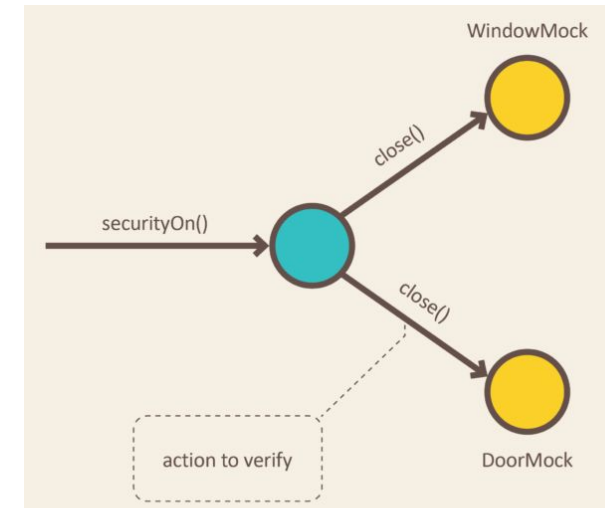
Modules & Test Double Variations

❖ Mock Object

- It Verifies the functions called, the call order, and the parameters passed from the CUT to the DOC
- It usually deals with a situation where multiple calls are made to it, and each call and response are potentially different

❖ Fake Object

- It provides a partial and simplified version of the production code
- It is used when the CUT depends on a component which makes testing difficult or slow



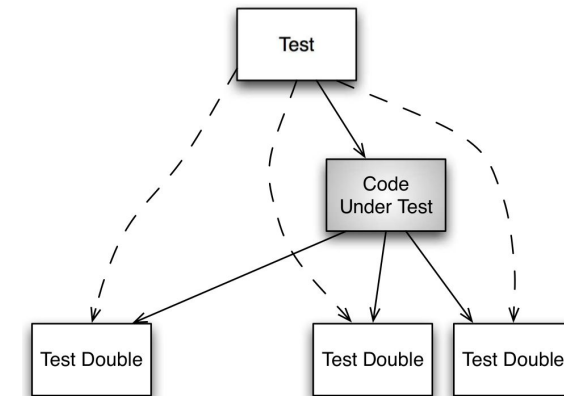
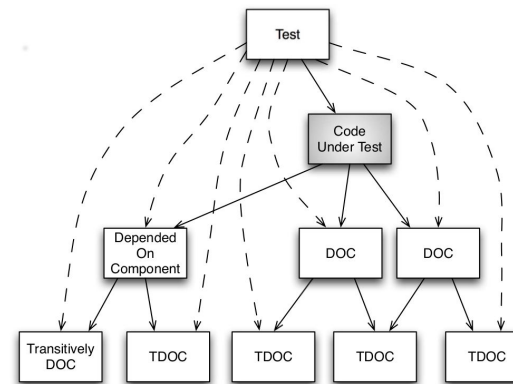
Faking in C

❖ Three primitive mechanisms to fake dependencies in C

- Preprocessor substitution
- Link-time substitution
- Function pointer substitution

❖ Link-time substitution

- Use it when you need to
 - Replace the DOC for the whole unit test,
 - Substitute a module where you do not control the interface
 - Eliminating dependencies on third-party libraries, hardware-dependent modules, or the OS
- Instead of linking the real module, we need to make a test double for the DOC and link it



Faking in C

❖ Function pointer substitution

- Use it when you want to replace the DOC for only some of the test cases
- You can use it everywhere that you control the interface
- It Uses some RAM, and compromises function declaration readability
- It allows great control over where and when functions get overridden

```
h module.h ×
function-pointer > module > h module.h > ...
1  #ifndef MODULE_H
2  #define MODULE_H
3
4  typedef void (*free_t)(void *);
5
6  void module_begin(free_t func);
7  void module_end(void);
8
9  #endif /* MODULE_H */
```

```
C module.c ×
function-pointer > module > C module.c > module_end(void)
1  #include <stdint.h>
2  #include <stdlib.h>
3  #include <module.h>
4
5  static uint8_t *ptr = NULL;
6  static free_t free_memory = NULL;
7
8  void module_begin(free_t func)
9  {
10     free_memory = (func == NULL) ? free : func;
11
12     module_end();
13
14     ptr = (uint8_t *)malloc(sizeof(uint8_t));
15 }
16
17 void module_end(void)
18 {
19     if (ptr != NULL)
20     {
21         free_memory(ptr);
22         ptr = NULL;
23     }
24 }
```

```
C test.c ×
function-pointer > C test.c > my_free(void *)
1  #include <stdio.h>
2  #include <unity.h>
3  #include <module.h>
4  #include <stdlib.h>
5
6  static void my_free(void *ptr)
7  {
8     printf("\nTest Code: my_free is called\n\n");
9     free(ptr);
10 }
11
12 void setUp()
13 {
14     module_begin(my_free);
15 }
16
17 void tearDown()
18 {
19     module_end();
20 }
21
22 void test_free_memory()
23 {
24     TEST_IGNORE();
25 }
26
27 int main(void)
28 {
29     UNITY_BEGIN();
30
31     RUN_TEST(test_free_memory);
32
33     return UNITY_END();
34 }
```

Faking in C

❖ Preprocessor substitution

- Use it when linker and function pointer substitutions can't do the job
- You can break a chain of unwanted includes with preprocessors
- You can also use it selectively or temporarily to override names

```
double > h double.h > ...
1  #ifndef DOUBLE_H
2  #define DOUBLE_H
3
4  #define putchar(x) _putchar(x)
5
6  int _putchar(int __c);
7
8  #endif /* DOUBLE_H */
9
```

```
double > C double.c > ...
1  #include <stdio.h>
2
3  /**
4   * Don't include double.h if you
5   * want to use the original putchar
6   */
7
8  extern int counter;
9
10 int _putchar(int __c)
11 {
12     counter++;
13     return putchar(__c);
14 }
15
```

```
C test.c > main(void)
1  #include <stdio.h>
2  #include "double.h"
3
4  int counter = 0;
5
6  int main(void)
7  {
8      putchar('a');
9      putchar('b');
10
11     printf("\n%d\n", counter);
12
13     return 0;
14 }
15
```

Faking in C

❖ Combination of link-time and function pointer substitutions

- We can combine link-time and function pointer substitution to work together
- A link-time stub can be created that contains a function pointer initialized to NULL
- A test case can override the NULL pointer and provide exactly the stub function
- You want the flexibility of the function pointers but not to change the interface of the DOC

```
C main.c
1 #include <stdio.h>
2 #include <module.h>
3
4 int main(void)
5 {
6     module_begin();
7     module_end();
8
9     printf("\nThe module in main.c\n\n");
10
11     return 0;
12 }
```

```
C test.c
1 #include <unity.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <module.h>
5 #include <double.h>
6
7 extern void (*free_memory)(void *);
8
9 void setUp() { module_begin(); free_memory = my_free; }
10
11 void tearDown() { module_end(); free_memory = free; }
12
13 void test_free_memory() { TEST_IGNORE(); }
14
15 int main()
16 {
17     UNITY_BEGIN();
18
19     RUN_TEST(test_free_memory);
20
21     return UNITY_END();
22 }
```

```
double.h
1 #ifndef DOUBLE_H
2 #define DOUBLE_H
3
4 void my_free(void *ptr);
5
6 #endif /* DOUBLE_H */
```

```
double.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <double.h>
4
5 void my_free(void *ptr)
6 {
7     printf("\nmy_free is called!\n");
8     free(ptr);
9 }
```

```
module.c
1 #include <module.h>
2 #include <stdint.h>
3 #include <stdlib.h>
4
5 static uint8_t *data_ptr = NULL;
6 void (*free_memory)(void *) = free;
7
8 void module_begin(void)
9 {
10     data_ptr = (uint8_t *)malloc(sizeof(uint8_t));
11 }
12
13 void module_end(void)
14 {
15     if (data_ptr != NULL)
16     {
17         free_memory(data_ptr);
18         data_ptr = NULL;
19     }
20 }
```

```
module.c
1 #include <module.h>
2 #include <stdint.h>
3 #include <stdlib.h>
4
5 static uint8_t *data_ptr = NULL;
6 void (*free_memory)(void *) = free;
7
8 void module_begin(void)
9 {
10     data_ptr = (uint8_t *)malloc(sizeof(uint8_t));
11 }
12
13 void module_end(void)
14 {
15     if (data_ptr != NULL)
16     {
17         free_memory(data_ptr);
18         data_ptr = NULL;
19     }
20 }
```


Dual-Targeting Strategy

- ❖ Dual-targeting means that the code is designed to run on at least two platforms:
 - The final target hardware
 - The development system
- ❖ Dual-targeting strategy is used because of:
 - The target hardware is expensive
 - Target hardware is not ready until late in the project
 - When target hardware is finally available, it may have bugs of its own
 - Long target build or upload times waste valuable time during the edit, compile, and test cycle
 - Compilers for the target hardware are considerably more expensive than native compilers
 - And etc.
- ❖ Concurrent hardware and software development is a reality for many projects

Dual-Targeting Strategy

❖ Benefits of Dual-Targeting

- It allows you to overcome the hardware restrictions
- Dual-targeting, like TDD, produces more modular and hardware independence designs

❖ Risks of Dual-Target Testing

- The risks are because of differences between the development and target environments
- Compilers may support different language features
- The runtime libraries may be different
- Primitive data types might have different sizes
- The target compiler may have bugs
- Byte ordering and data structure alignments may be different

Dual Targeting & Link-Time Faking Techniques

- ❖ If you want to test your module **ONLY** on your **PC**
 - Make a test double using the link-time substitution technique for Arduino.
 - The test double shall have Arduino.h and optionally Arduino.cpp to implement the fake functions
 - In this technique your modules will be dependent of Arduino
 - We have to include Arduino.h in modules and directly use Arduino functions
- ❖ If you want to follow the dual targeting strategy and make portable modules
 - Make an abstraction layer over Arduino in order to make your modules independent of Arduino. In this way your modules don't directly depend on Arduino.
 - Use the link-time faking technique to make test doubles for the modules in the abstraction layer
 - In the modules of the abstraction layer you can use Arduino functions and macros

Dual Targeting & Function Pointer Faking Techniques

- ❖ If you want to test your module on **ONLY** your **PC**
 - Generally avoid using the function pointer substitution technique to make test doubles.
 - The readability of your code is decreased.
- ❖ If you want to follow dual targeting and test your module on your **PC** and the **µC** without needing to create an abstraction layer over Arduino, use the function pointer technique. Even if you need to abstract a module or a framework, you can use the function pointer technique to make test doubles for the abstracted modules.
- ❖ Avoid using the preprocessor technique to make test doubles. It affects the CUT
 - But preprocessor substitution can be useful when we only want to change function names

Code Coverage

❖ What is Code Coverage?

- Is the percentage of the code which is covered by automated tests
- Determines which parts of the code is executed through a test run, and which parts is not

❖ Why Measure Code Coverage?

- To know how well our tests actually test the production code
- To know whether we have enough testing in place
- To maintain the test quality over the lifecycle of the project

❖ Types of Coverage measured

- **Statement coverage:** measures whether each statement is executed
- **Condition coverage:** measures whether each sub-expression evaluated both to true and false
- **Branch coverage:** measures which branches (e.g. if-else) in flow control structures are followed
- **Function coverage:** measures whether each function in the program been called

Code Coverage

- ❖ **Code instrumentation** refers to insert code in programs to monitor
 - Code tracing, debugging, analyzing, performance measuring, data logging and etc.
- ❖ Mainly there are three code Instrumentation approaches
 - **Source-code** instrumentation: Add the instrumentation code to the source code and compile the it with the compilation tool chain to produce an instrumented program.
 - Is done by developers and processed in the preprocessor step.
 - **Compile-time** instrumentation: Is done during compilation of source files, and the instrumentation is performed by the compiler.
 - E.g. debugging(-g), profiling(-pg), code coverage(--coverage), and etc.
 - **Runtime** instrumentation: The instrumentation code and data are inserted into the executable binary files during execution. It is done during runtime.
 - E.g. Dynamic binary instrumentation tools like [Intel Pin Tool](#)

Code Coverage in C

- ❖ We need to use **gcov** to instrument coverage code in files
 - **gcov** is the code coverage library for **gcc/g++**.
 - Compile source files with **--coverage**, and link against **-lgcov**
 - For **.cpp** projects, you can use **g++** compiler or link **-lstdc++** if you use **gcc**.
 - Run the test to produce **.gcda** and **.gcno** files
- ❖ To generate code coverage reports:
 - On Linux you can use **lcov**(a graphical front-end for gcov) and **genhtml**(html report generator).
 - Or you can use **gcovr**(a python package). For more info about gcovr look at the [documentation](#)
 - Run **gcovr -r <root-directory> --html-details -o <output-directory>/index.html**
 - Specify root-directory and the output-directory of your project. Look at the [options of gcovr](#)
- ❖ Example
 - `mkdir -p build; gcc --coverage ./test/test.c ./module/module.c -lunity -lgcov -o ./build/test;`
 - `./build/test; gcovr -r . --html-details -o build/index.html`



Modularity and Software Testing - Exercises

- ❖ Exercise 12: Develop a multiple-instance **stack** module
 - Size of instances are set during runtime when we create them.
 - The module shall be fully tested using the TDD technique on **YOUR PC** and **ESP32**
 - To make a test double for the dependency use the function pointer faking technique

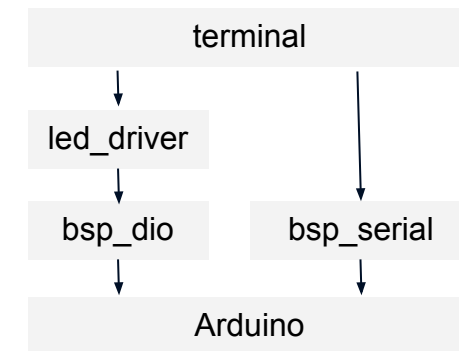
- ❖ Exercise 13: Develop a multiple-instance module (**mmetro**) like the **Metro** library
 - The module shall be fully tested using the TDD technique on **YOUR PC** and **Teensy**
 - The module shall be able to measure microseconds and milliseconds
 - Using **millis()** and **micros()** in Arduino
 - To make a test double for the dependency use the function pointer faking technique

Modularity and Software Testing - Exercises

- ❖ Connect your LDR series with a $4.7\text{k}\Omega$ resistor to an analog pin and a LED series with 120Ω to a PWM pin on Teensy. Specify a range of resistance for light intensity in your room
 - E.g. for max. light the LDR resistance is $2\text{k}\Omega$ and for min. light it is $50\text{k}\Omega$
- ❖ Make a driver module (**ldr_driver**) for the LDR to read the light intensity in percentage
 - Means that the max. resistance is equivalent to 0% and max. resistance is equivalent to 100%
- ❖ Make a driver module (**led_driver**) for a LED to change its brightness in percentage
 - You need to use PWM. Means that 0% is equivalent to 0 and 100% of brightness is equivalent to 255.
- ❖ Make a program to read the light intensity in percentage every 100ms and apply it to the LED.
 - For timing use an **interval timer**.
- ❖ Use the **TDD** technique to test your modules. (Code coverage shall be 100%)
- ❖ **Exercise 14:** Use the link-time faking technique to develop the modules on your computer
- ❖ **Exercise 15:** Follow the **dual targeting strategy**. Test your modules on Teensy.
 - Create an abstraction layer over Arduino to make your modules independent of Arduino.

Modularity and Software Testing - Exercises

- ❖ **Exercise 16:** Make a program for ESP32 to control the built-in LED in a terminal.
- ❖ The program shall have a menu system with two options to turn the LED on and off
- ❖ You shall develop a driver module(led_driver) for the LED
- ❖ You shall develop an application module(terminal) for user interaction.
- ❖ Your modules shall be independent of Arduino
 - Create an abstraction layer over Arduino(bsp_dio and bsp_serial modules)
- ❖ Use TDD and dual targeting strategy to test your modules.
 - Develop the modules on your computer.
 - The code coverage for your tests shall be 100%
- ❖ Do the integration testing.
- ❖ Use the link-time faking technique to make test doubles for the dependencies



Modularity and Software Testing

❖ Some useful links

- [Modular Programming in C](#)
- [Modular Approach in Programming](#)
- [Effective Tests: Test Doubles](#)
- [Test Doubles — Fakes, Mocks and Stubs](#)
- [Test Double](#)
- [Linker Substitution in C](#)
- [Function pointers vs. Preprocessor vs. Link-time substitution](#)
- [SOLID Agile Development](#)
- [SOLID Principles made easy](#)
- [SOLID Principle in Programming](#)
- [SOLID Principles: Explanation and examples](#)
- [gcovr documentation](#)
- [About Code Coverage](#)
- [Instrumentation \(computer programming\)](#)
- [The Ultimate List of Code Coverage Tools](#)
- [Code Coverage Using GNU Gcov and Lcov](#)
- [Code Coverage in C with gcov and lcov](#)
- [Using gcov and lcov](#)
- [Code instrumentation](#)