presented by

UEFI Development Anti-Patterns

Spring 2017 UEFI Seminar and Plugfest
March 27 - 31, 2017
Presented by Chris Stewart (HP Inc.) Lead Security Developer, Firmware

Agenda

- Design Patterns
- Anti-Patterns
- Anti-Patterns Categories
  - Haste
  - Apathy
  - Narrow-mindedness
  - Sloth
  - Avarice
  - Ignorance
  - Pride

Design Patterns

Design Patterns

- A *Design Pattern* is a standard, reusable design.
- It is not a data structure but a generality about types of data structures.
  - Not a Linked List but an *Iteration* construct applicable to *any* collection.
  - Not an implementation but a reusable *interface* type, applicable to multiple implementations.
- PPI or Protocol
  - Reusable container
  - Discoverable
  - Common registration interface
  - Does not define implementation *content*
  - Defines an implementation *approach*

Design Patterns

A few design patterns:

- **Singleton object**: Only one instance of a particular object is allowed to be instantiated.
  - Many UEFI protocols are singleton objects because there is exactly one such protocol instance in the firmware. *LocateProtocol* finds the single instance.
  - Other UEFI protocols are *not* singleton objects because multiple instances can implement the protocol's defined interface. *HandleProtocol* finds the various instances, and the caller can iterate through the collection to find and use one or more particular instances.
- **Composite**: Zero to many object instances encapsulated and treated in the same way.
  - UEFI protocol registration represents a composite registration mechanism.
  - All protocols register the same way.
  - All protocols are discovered in one or more UEFI-defined ways.
  - All protocols handle events in the same way.
- **Iterator**: Exposes methods for sequential access to the elements of the collection. Iterators hide the means of implementing the collection.
  - UEFI lacks an iterator design pattern
  - EFI_LIST_FOR_EACH (MdeModulePkg) isn't really an iterator but a C language helper macro.
  - UEFI Red Black trees use a different iteration construct. No design-level commonality.

Anti-Patterns

Anti-Patterns

- 1998: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*
  - William J. Brown
  - Raphael C. Malveau
  - Hays W. McCormick
  - Thomas J. Mowbray
- Identifies software development patterns that lead to failure rather than success.
- Further identifies software development processes that commonly lead to project failure.
- Goal is to identify these patterns of development and root them out.

Anti-Patterns

A few Anti-Patterns

- **Dead Code**: Unused code that still exists in a project build, even though there are no code paths for that code.
- **Poltergeists**: *Living Dead Code*. Code that nobody understands, but removing it causes build errors and introduces bugs. Therefore, everybody leaves it alone.
- **Functional Decomposition**: A single “main” routine handles everything from interface entry all the way down to implementation.
- **Golden Hammer**: Attempting to apply the same solution to every development problem. This includes the application of a good Design Pattern in the wrong problem space.
- **Dead End**: Adding your own modifications to a vendor-provided source code module.

AntiPatterns Categories

- Haste
- Apathy
- Narrow-mindedness
- Sloth [Expedience]
- Avarice [Greed]
- Ignorance
- Pride

From *AntiPatterns*, by William Brown, Raphael Malveau, Skip McCormick, and Tom Mowbray

Haste

```
EFI_STATUS EFIAPI GeeWizFeature(IN OUT GeeWizAppData_t *AppData)
{
   EFI_STATUS          Status = EFI_NOT_STARTED;
   GeeWizDeviceData_t  *Device = NULL;
   GeeWizDevProtocol_t *Proto = NULL;

   Status = VerifyAppDataOnInput(AppData);
   if(EFI_SUCCESS == Status)
   {
      Status = gBS->LocateProtocol(&gGeeWizDeviceProtocolGuid, NULL,
                                   (void **)&Proto);
   }
   if((EFI_SUCCESS == Status) && (NULL != Proto))
   {
      Status = Proto->ApplyAppData(AppData);
   }

   return Status;
}
```

Haste

```
EFI_STATUS EFIAPI GeeWizFeature(IN OUT GeeWizAppData_t *AppData)
{
    EFI_STATUS          Status = EFI_NOT_STARTED;
    GeeWizDeviceData_t  *Device = NULL;
    GeeWizDevProtocol_t *Proto = NULL;

    Status = VerifyAppDataOnInput(AppData);
    if(EFI_SUCCESS == Status)
    {
        // Device team is late with gGeeWizDeviceProtocolGuid implementation.
        // Status = gBS->LocateProtocol(&gGeeWizDeviceProtocolGuid, NULL, (void **)&Proto);
    }
    // TO-DO: Uncomment when Device team is ready.
    // if((EFI_SUCCESS == Status) && (NULL != Proto))
    // {
    //     Status = Proto->ApplyAppData(AppData);
    // }
    return Status;
}
```

Late

Dead Code

Dead Code

Haste Alternative

GeeWizDevProtocol_t Definition

```
typedef struct _GeeWizDevProtocol_t {
   UINT64         Signature;
   ApplyAppData_t ApplyAppData;
} GeeWizDevProtocol_t;

#define TEST_MODE SIGNATURE_64(‘T’, ‘E’, ‘S’, ‘T’, ‘M’, ‘O’, ‘D’, ‘E’)
#define PRODUCTION_MODE SIGNATURE_64(‘P’, ‘R’, ‘O’, ‘D’, ‘M’, ‘O’, ‘D’, ‘E’)

#define EFI_TEST_MODE EFIERR(1037)

// ALLOW_TEST_MODE is defined via  <BuildOptions> in DSC file, e.g.
 <BuildOptions>
          MSFT:*_*_*_CC_FLAGS = /D ALLOW_TEST_MODE=1
          GCC:*_*_*_CC_FLAGS = -D ALLOW_TEST_MODE=1
//
// OR Globally
 [BuildOptions]
          MSFT:*_*_*_CC_FLAGS = /D ALLOW_TEST_MODE=1 ...
          GCC:*_*_*_CC_FLAGS = -D ALLOW_TEST_MODE=1 ...
```

Haste Alternative

Implement simple unit testing where possible.

```
EFI_STATUS EFIAPI GeeWizFeature(IN OUT GeeWizAppData_t *AppData)
{  <SNIP out declarations for readability purpose only>
   Status = VerifyAppDataOnInput(AppData);
   if(EFI_SUCCESS == Status)
   {
      Status = FindOrInstallGeeWizDeviceProtocol(&Proto);
   }
   if((EFI_SUCCESS == Status) && (NULL != Proto))
   {
      Status = Proto->ApplyAppData(AppData);
      if((EFI_SUCCESS == Status) && (Proto->Signature == TEST_MODE) && (IsOfficialBuild()))
      {
         Status = EFI_TEST_MODE; // Or just use a status like EFI_UNSUPPORTED
      }
   }

   return Status;
}
```

Haste Alternative

Implement simple unit testing where possible.

```
EFI_STATUS EFIAPI FindOrInstallGeeWizDeviceProtocol(OUT GeeWizDevProtocol_t **Proto)
{
   EFI_STATUS Status = EFI_NOT_STARTED;
   if((NULL == Proto) || (NULL != *Proto))
   {
      Status = EFI_INVALID_PARAMETER;
   }
   else
   {
      Status = gBS->LocateProtocol(&gGeeWizDeviceProtocolGuid, NULL, (void **)&Proto);
      // On Success Proto->Signature == PRODUCTION_MODE;
      if(EFI_SUCCESS != Status)
      {
         Status = InstallTestModeGeeWizDeviceProto(Proto);
         // On Success Proto->Signature == TEST_MODE;
      }
   }

   return Status;
}
```

Haste Alternative

Implement simple unit testing where possible.

```
EFI_STATUS EFIAPI FindOrInstallGeeWizDeviceProtocol(OUT GeeWizDevProtocol_t **Proto)
{
   EFI_STATUS Status = EFI_NOT_STARTED;
   if((NULL == Proto) || (NULL != *Proto))
   {
      Status = EFI_INVALID_PARAMETER;
   }
   else
   {
      Status = gBS->LocateProtocol(&gGeeWizDeviceProtocolGuid, NULL, (void **)&Proto);
#if ALLOW_TEST_MODE
      // On Success Proto->Signature == PRODUCTION_MODE;
      if(EFI_SUCCESS != Status)
      {
         Status = InstallTestModeGeeWizDeviceProto(Proto);
         // On Success Proto->Signature == TEST_MODE;
      }
#endif
   }

   return Status;
}
```

Haste Alternative

FDF File Example

```
!if $(ALLOW_TEST_MODE) == TRUE
  INF DriverPath\GeeWizDeviceDriver\GeeWizDeviceDriverStub.inf
!else
  INF DriverPath\GeeWizDeviceDriver\GeeWizDeviceDriver.inf
!endif

# --------------------------------------------------------------------------
# You might also define different ALLOW TEST MODE variables to get more granularity.
# This allows for easy integration into builds like OVMF, which might never
# Have a GeeWiz device to speak of.
# --------------------------------------------------------------------------

!if $(ALLOW_GEEWIZ_TEST_MODE) == TRUE
  INF DriverPath\GeeWizDeviceDriver\GeeWizDeviceDriverStub.inf
!else
  INF DriverPath\GeeWizDeviceDriver\GeeWizDeviceDriver.inf
!endif
```

Eliminate Haste

- Make code reviews a priority, even if you have to push out schedule.
- Attend to code review input even though doing so is time consuming.
- Make sure that security threat analyses are a priority.
- Pay attention to ad-hoc bug reports even if the test case isn't in the test plan.
- Refactor copy-n-paste code blocks into shared libraries.
- Make sure to use static analysis tools and to attend to the static analysis results.

Apathy

- Many UEFI development projects integrate third-party code.
- All UEFI projects integrate open source code.
- Often multiple internal teams are responsible for different areas of UEFI development.
- It is often easy to place blame on one of these two code sources.

Apathy

Some code for thought:

```
EFI_STATUS EFIAPI DriverEntry(IN EFI_PEI_FILE_HANDLE FileHandle, IN CONST
EFI_PEI_SERVICES **PeiServices)
{
   EFI_STATUS Status = EFI_NOT_STARTED;
   MyPtr_t    *Ptr = NULL;

   Ptr = GetInputFromThirdPartyDevice();
   ASSERT(Ptr != NULL); // Boot hangs when P
   Status = InitializeDriver(Ptr);
   ASSERT(!EFI_ERROR(Status)); // Bad
   // Rest of driver goes here, including PPI installation.

   // Boot hangs if this driver doesn't load
   // Must Return EFI_SUCCESS
   return EFI_SUCCESS;
}
```

Assert Compiled Out of Release Code

Assert Compiled Out of Release Code

Incorrect Driver Status Returned

Apathy

Better. Drivers need to deal with runtime problems.

```
EFI_STATUS EFIAPI DriverEntry(IN EFI_PEI_FILE_HANDLE FileHandle, IN CONST EFI_PEI_SERVICES **PeiServices)
{
   EFI_STATUS Status = EFI_NOT_STARTED;
   MyPtr_t    *Ptr = NULL;
   Ptr = GetInputFromThirdPartyDevice();
   if(Ptr == NULL)
   {
      PostPoneDriverInitialization();
      Status = EFI_SUCCESS;
   }
   else
   { // Driver needs to handle unexpected inputs. BIOS should be OK with or without driver
      Status = InitializeDriver(Ptr);
      if(Status != EFI_SUCCESS)
      {
         Status = HandleNonCompliantDevice(Ptr);
      }
   }
   if(Status == EFI_SUCCESS)
   {
      // Rest of driver goes here, including PPI installation.
   }
   return Status;
}
```

Narrow-mindedness

UEFI is designed for *runtime binding* but too often we force *build time binding* instead.

```
EFI_STATUS EFIAPI EntryPoint(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
   EFI_STATUS Status = EFI_NOT_STARTED;
#if VENDOR_A_PLATFORM
   VendorAProtocol_t *Proto = NULL;
#else
   VendorBProtocol_t *Proto = NULL;
#endif

#if VENDOR_A_PLATFORM
   Status = gBS->LocateProtocol(&gVendorAProtocolGuid, NULL, (void **)&Proto);
#else
   Status = gBS->LocateProtocol(&gVendorBProtocolGuid, NULL, (void **)&Proto);
#endif

   // Now use the protocol

   return Status;
}
```

Vendor A / Vendor B Split Must Continue Here

Narrow-mindedness

This is better.

```
EFI_STATUS EFIAPI EntryPoint(IN EFI_HANDLE ImageHandle, IN
EFI_SYSTEM_TABLE *SystemTable)
{
   EFI_STATUS Status = EFI_NOT_STARTED;
   void       *ProtoA = NULL;
   void       *ProtoB = NULL;

   gBS->LocateProtocol(&gVendorAProtocolGuid, NULL, &ProtoA);
   gBS->LocateProtocol(&gVendorBProtocolGuid, NULL, &ProtoB);

   // Now use the protocol
   Status = HandleMultiVendorProtocol(ProtoA, ProtoB);

   return Status;
}
```

Narrow-mindedness

This is better.

```
EFI_STATUS HandleMultiVendorProtocol(IN void *ProtoA, IN void *ProtoB)
{
   EFI_STATUS Status = EFI_NOT_STARTED;

   if(NULL != ProtoA)
   {
      Status = HandleProtocolA ((VendorAProtocol_t *)ProtoA);
   }
   else
   {
      Status = HandleProtocolB ((VendorBProtocol_t *)ProtoB);
   }

   return Status;
}

EFI_STATUS HandleProtocolA(VendorAProtocol_t *Proto)
{
   // Implement
}

EFI_STATUS HandleProtocolB(VendorBProtocol_t *Proto)
{
   // Implement
}
```

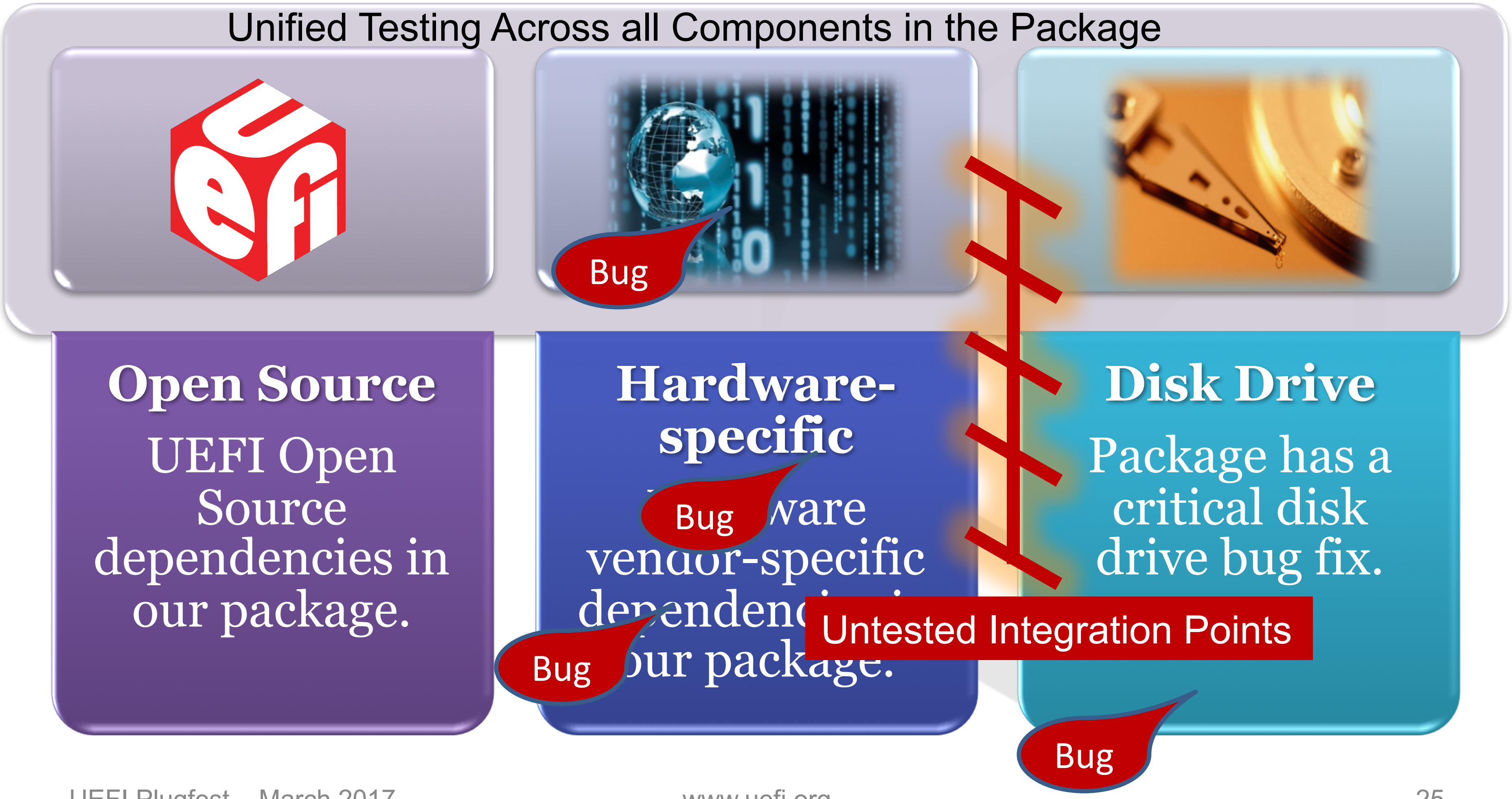
Sloth

We can’t take that risk.

- Cross-team development
  - Desire to avoid risk by only taking small code changes.
  - Larger code changes are perceived as larger risks, even though they have been tested in an integrated way.
  - This is human nature, but it leads to problems.

Sloth

I don't want the whole thing. Too risky!


Unified Testing Across all Components in the Package
Bug
Bug
Bug
Untested Integration Points
Bug

Open Source

UEFI Open Source dependencies in our package.

Hardware-specific

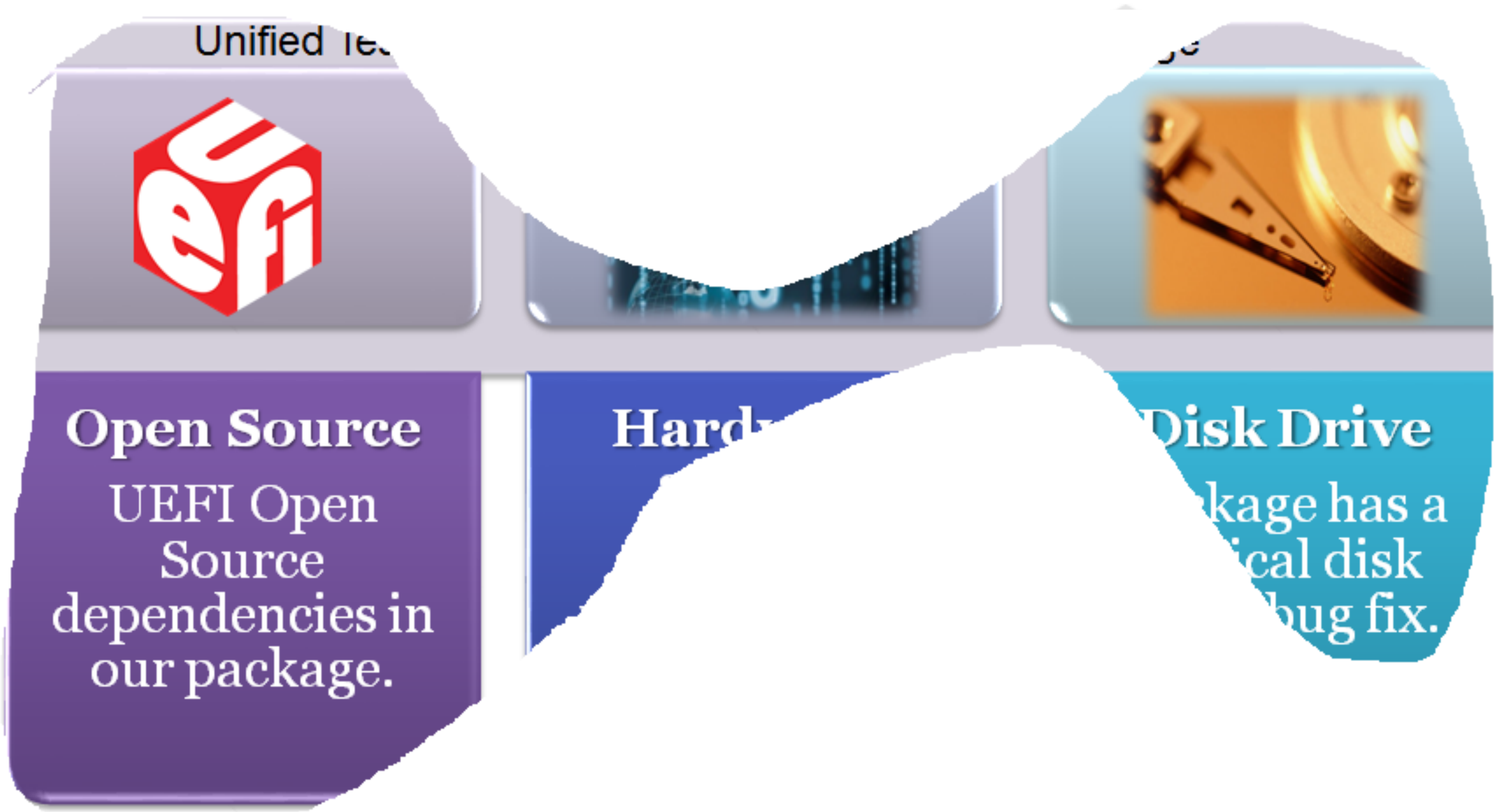
ware vendor-specific dependenc our package.

Disk Drive

Package has a critical disk drive bug fix.

Sloth

This is all I want. It fixes my bug.



Unified Te

Open Source

UEFI Open Source dependencies in our package.

Hard

Disk Drive

kage has a
ical disk
bug fix.

Sloth Alternative

GeeWizDevProtocol_t Build Rules in DSC file

```
[Components.X64]
 Universal\Devices\GeeWizDevice\GeeWizDxe.inf {
    <LibraryClasses>
!if OFFICIAL_RELEASE_BUILD
      GeeWizImplLib|OurDevicePkg\Library\GeeWizLib\GeeWizDeviceLib.inf
!elseif UNIT_TEST_NEGATIVE_VALUES_BUILD
      GeeWizImplLib|UnitTestPkg\Library\GeeWizLib\GeeWizFailureLib.inf
!elseif UNIT_TEST_POSITIVE_VALUES_BUILD
      GeeWizImplLib|UnitTestPkg\Library\GeeWizLib\GeeWizSuccesLib.inf
!elseif INTEGRATION_TEST_BUILD
      GeeWizImplLib|TestDevicePkg\Library\GeeWizLib\GeeWizTestDeviceLib.inf
!elseif VIRTUAL_ENVIRONMENT_BUILD
      GeeWizImplLib|VirtualDevicePkg\Library\GeeWizLib\GeeWizVirtualDeviceLib.inf
!endif
  }
```

Avarice

Dead End Design Anti-Pattern

- Vendor provides a large collection of modules.
- As recipients of the vendor code, we make inline modifications to reflect the specific needs of our own products.
- Every time we get vendor bug fixes, we have to merge in all of our changes.

Better Alternative

- Add hooks to the vendor code that call out to our own modules.
- The merge problem is greatly reduced.
- Quicker import of new vendor code and fewer bugs.

Avarice

```
EFI_STATUS EFIAPI VendorFeature(IN OUT VendorAppData_t *AppData)
{
   EFI_STATUS          Status = EFI_NOT_STARTED;
   VendorType_t        *VendorPtr = NULL;
   // Lots more vendor declarations

   Status = VerifyVendorAppDataOnInput(AppData);
   if(EFI_SUCCESS == Status)
   {
      Status = gBS->LocateProtocol(&gGeeWizDeviceProtocolGuid, NULL, (void **)&Proto);
      // Vendor specific processing is also here.
   }
   if((EFI_SUCCESS == Status) && (NULL != Proto))
   {
      Status = Proto->ApplyAppData(AppData);
   }

   // More vendor specific processing is here.
   if(VendorPtr->VendorData == GEE_WIZ_TRIGGER)
   {
      Status = Proto->MoreGeeWizProcessing(VendorPtr);
      // More vendor specific processing is also here.
   }
   return Status;
}
```

Avarice

A Better Alternative

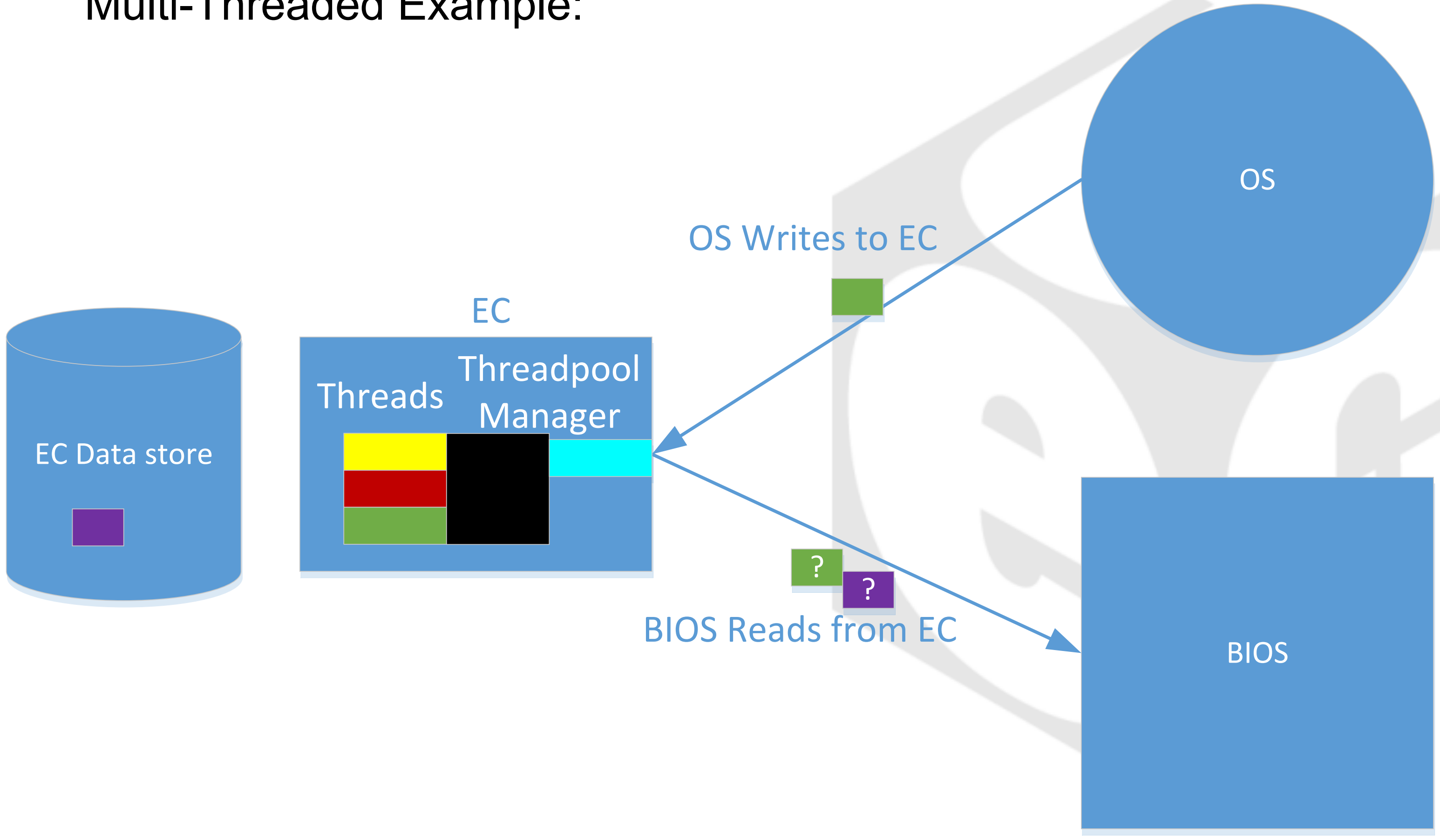
```
EFI_STATUS EFIAPI VendorFeature(IN OUT VendorAppData_t *AppData)
{
   EFI_STATUS          Status = EFI_NOT_STARTED;
   VendorType_t        *VendorPtr = NULL;
   // Lots more vendor declarations

   Status = VerifyVendorAppDataOnInput(AppData);
   if(EFI_SUCCESS == Status)
   {
      Status = PreProcessVendorFeatureHook(AppData);
      // Vendor specific processing is also here.
   }

   // More vendor specific processing is here.

   Status = PostProcessVendorFeatureHook(VendorPtr);

   return Status;
}
```

Ignorance

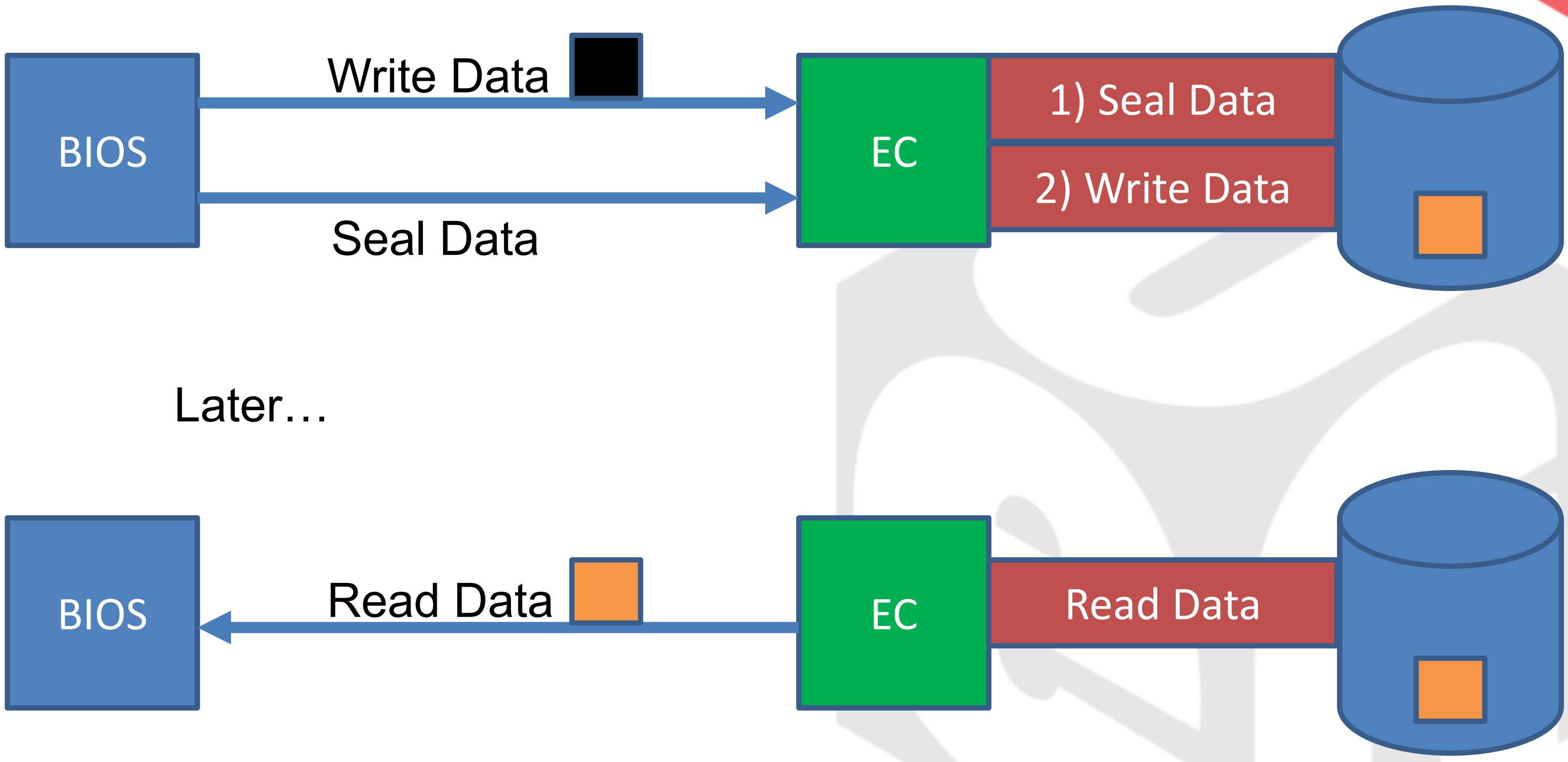
- Multi-threaded subsystems
- Embedded Controllers (EC) are multi-threaded today.
- UEFI must deal with the fact that the EC will implement multi-threaded Design Patterns

Ignorance

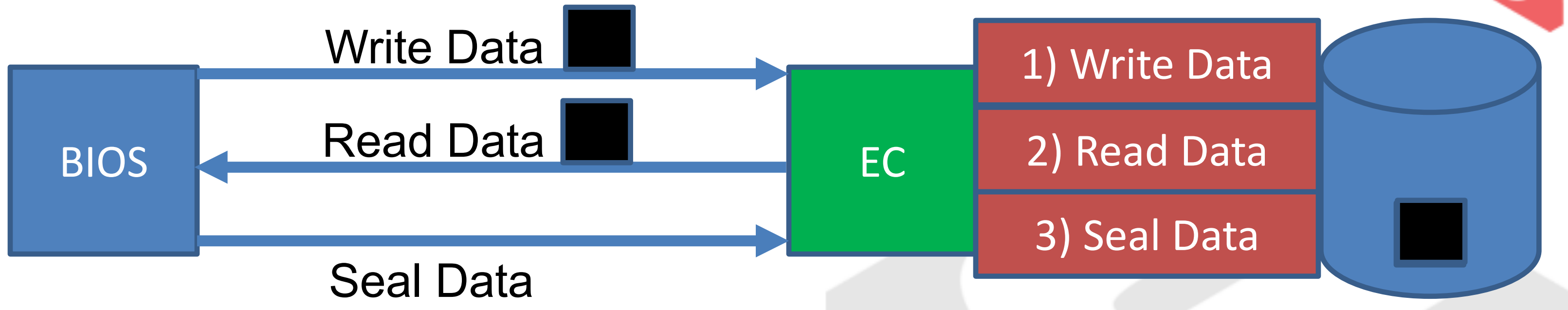
Multi-Threaded Example:



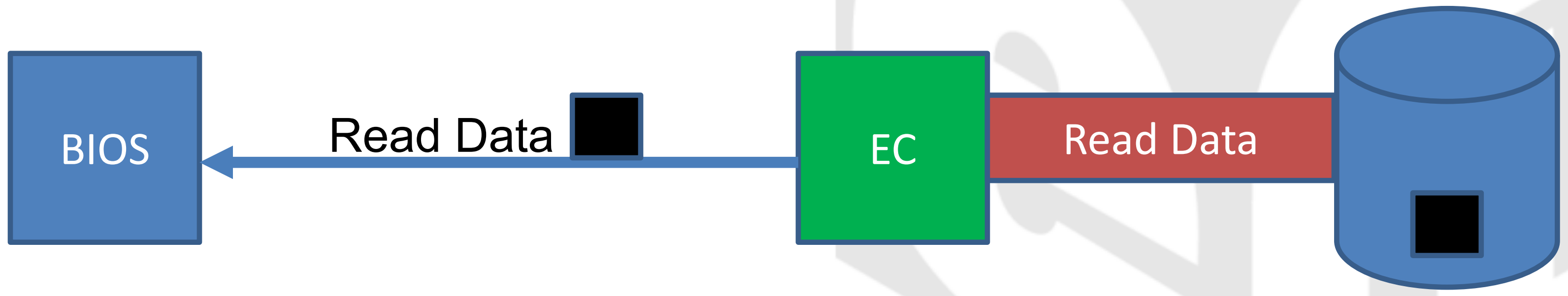
Ignorance: Example



Ignorance: Better



Later…



Pride

I’m a low-level coder.

- With C code in general, there’s nothing to stop you from looking at low-level data.

```
Data = IoRead8(0x73A297);
if(Data & ~(1 << 6)) …
if(Data | (1 << 6)) …
```

Pride

- Maybe we improve on this somewhat

```
Data = IoRead8(VENDOR_COOL_DATA_ADDR); // See page 674 of
vendor documentation for details
if(Data & ~(BIT6)) … // See page 1,273 of vendor
documentation for details
if(Data | (BIT6)) … // See page 1,274 of vendor
documentation for details. There is a note at the top.
Data = ReadUsbFeatureCapabilities();
if(IsUsbCool(Data)) { // Use the USB cool feature }
if(!IsUsbCool(Data))
{
   // write audit entry that USB cool feature not enabled
}
```

Address the Anti-Patterns

- **Haste**: Attend to those time-consuming activities that always improve code quality.
- **Apathy**: Turn integration challenges into unit-testing opportunities.
- **Narrow-mindedness**: Make your code work with *any* vendor's code.
- **Sloth**: Build trust among disparate teams.
- **Avarice**: Design your code to be ready to integrate anywhere.
- **Ignorance**: Study the other sub-systems.
- **Pride**: Write your code to make it easy to hand off to another.

Thanks for attending the Spring 2017 UEFI Seminar and Plugfest

For more information on the UEFI Forum and UEFI Specifications, visit http://www.uefi.org

presented by