International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:7, 2008

# Energy Conscious Builder Design Pattern with C# and Intermediate Language

Kayun Chantarasathaporn, and Chonawat Srisa-an

***Abstract*—**Design Patterns have gained more and more acceptances since their emerging in software development world last decade and become another de facto standard of essential knowledge for Object-Oriented Programming developers nowadays.

Their target usage, from the beginning, was for regular computers, so, minimizing power consumption had never been a concern. However, in this decade, demands of more complicated software for running on mobile devices has grown rapidly as the much higher performance portable gadgets have been supplied to the market continuously. To get along with time to market that is business reason, the section of software development for power conscious, battery, devices has shifted itself from using specific low-level languages to higher level ones. Currently, complicated software running on mobile devices are often developed by high level languages those support OOP concepts. These cause the trend of embracing Design Patterns to mobile world.

However, using Design Patterns directly in software development for power conscious systems is not recommended because they were not originally designed for such environment. This paper demonstrates the adapted Design Pattern for power limitation system. Because there are numerous original design patterns, it is not possible to mention the whole at once. So, this paper focuses only in creating Energy Conscious version of existing regular "Builder Pattern" to be appropriated for developing low power consumption software.

***Keywords*—**Design Patterns, Builder Pattern, Low Power Consumption, Object Oriented Programming, Power Conscious System, Software.

## I. Introduction

IN this era, mobile devices gain much more popular from so many supportive reasons such as lower price and better communication infrastructure. However, when mentioning about mobile equipments, one of the major issues we have to concern is the battery life. This is why battery-powered equipments are sometimes called power conscious system or PCS. Though manufacturers and researchers have tried to develop various technologies in both hardware and software to optimize energy from battery for these movable gadgets, to scope the research, this paper pays attention only in software section.

At first, software for handheld equipments usually created by specific low-level language, such as assembly, but later higher level languages such as C became more popular. One of the factors is the marketing reason that needs shorter development stage for faster launch time. Though everyone accepts that software created by low-level language can work fast, its development period is usually too long when compared with one created by higher level language. Another reason that made high level language gain more popular is its much easier reusability. Currently, software running on mobile devices is much more complicated than one in last decade but can launch faster because of this reason.

Since 1990, trend of commercial software development has shifted from procedural programming to object oriented programming (OOP) as the latter is rather more appropriated in developing complex application. OOP is also designed for well support in reusability. From this reason, there have been many guides for reusing existing good solutions to the problems. The outstanding one is Design Patterns.

Design Patterns were the collections of existing useful solutions in OOP software development. It has been well-known since early 1990s in the presentation in Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conferences by Gang of Four [1]. Later, this Gang issued a famous book that collected 23 fundamental design patterns which had been considered as another de facto standard of OOP professional development.

Design Patterns divided existing patterns to 3 groups [2]. First, Creational Patterns those concerned about how objects were created. Second, Structural Patterns which interested in putting objects together in to a larger structure. And, third, Behavioral Patterns those focused in collaborations among objects to achieve a particular goal.

Design Patterns have been very useful when applied with software creation in general environment on regular computers. However, for mobile systems which are power concerned environment due to limited life of battery, some patterns are not quite immediately appropriated because they were originally not designed for this scarce power situation.

This paper intends to solve this constraint by introducing Energy Conscious Design Patterns. EC Design Patterns are based on existing patterns in GOF book. However, since there are so many patterns, it is not possible to mention all patterns in 1 article. So, at first, we decided to introduce one of famous patterns in Creational Patterns for energy conscious environment that is EC Builder pattern.

The details in this article are as follows. Former studies of Low Power Consumption Software are introduced. Then, the

Kayun Chantarasathaporn is a Ph.D. student in Faculty of Information Technology, Rangsit University, Muang, Pathumtani, 12000, Thailand (e-mail:kayun@kayun.com).

Chonawat Srisa-an is the assistant professor in Faculty of Information Technolgy, Rangsit University, Muang, Pathumtani, 12000, Thailand (e-mail: chonawat@rangsit.rsu.ac.th).

factors those affect power conscious systems and software writing strategies in C# for this environment. There will be some mentions about Design Patterns in general. The overview of Builder pattern will be pointed. Next, some techniques for seeking EC Builder pattern will be described. There are sample codes of EC Builder pattern written in C# and the intermediate language (IL) generated from it. The paper will show the result of experiments about energy usage comparison between regular and EC Builder pattern.

This paper uses term power and energy interchangeably as Tiwari did [3] except in calculation part.

## II. Low Powered Consumption Software Study

Up to now, there have been quite numerous research articles pointing at software and its energy consumption. However, they can be classified to just a few scopes, such as, power analysis at low level language, compilation techniques those can create energy optimized codes, strategies for creation and implementation of software for power concerned system, boundary of usage time in embedded software, tools that help automatically find power critical points, and comparison of energy needed among different writing styles at the layer of high level language. The samples of researches just mentioned are as follows.

A well-known article [3] which is considered as the first research in the field of low power consumption in the software viewpoint is one from Tiwari and his team. They studied the power consumption of each major assembly command for specific CPU, 486DX2-S and the reasons in low level of software those affect power desire, such as, inter-instruction and cache miss effects. Though everyone had known that different commands should need different level of power, this Tiwari's work clarified how much they were.

Tiwari also recommended compilation techniques for the focus of low energy in another article [4]. He pointed out that the compiler should reorder the instructions to reduce switching since this activity required more power. Also, the code generated from compiler should choose using register instead of memory when possible since the registers use less

Studying about the time boundary used in software was done by Li and Malik [5]. They tried to find the time scope and critical points of software implementation with the help of linear programming techniques those applied to the high level language source code written in C.

Seeking automatic tool that can help code optimization was studied by Peymandoust et al [6]. Usually, in embedded system, software should be optimized as much as possible to consume less power. However, in the past, this process was done manually. Peymandoust used Profiler to help in finding critical points in term of basic blocks and proposed Symsoft which aimed automatically find some way that could produce acceptable outputs from the same input while using less power.

There was also a study of comparisons in term of power consumption and performance between Object-Oriented and Procedural coding style [7]. The result was as expected that OOP consumed more resources than procedural one. However, the study demonstrated that this should be acceptable when compared with the benefits gaining from development in OOP style, such as, reusability, member private management, etc.

## III. C# Software Writing Strategies for PCS

Though so many times that the outcome of codes written in high level languages are similar, the power consumption of each solution is different noticeably. One of the reasons is power needed for each detail instruction might not be the same. From our prior research [8], we found that it was true and we recommended strategies for writing codes in OOP (with C#) that could lessen required energy.

The summary of recommendation is in Table I.

TABLE I
SUMMARY OF C# CODING STRATEGIES FOR PCS

| What to work with | Choice 1 | Choice 2 | Recommend |
|---|---|---|---|
| Group of attributes creation | class | struct | **struct** |
| Field | static | dynamic | **static** |
| Method | static | dynamic | **static** |

## IV. Design Patterns

As mentioned in [9], design pattern, in software engineering viewpoint, is a general repeatable solution to a common happening problem in software architecture. Design pattern is not a finished design that can be transformed directly into code. However, it is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Though the goal of algorithm and design pattern is quite similar in trying to solve problems, their details are different, such as, algorithm try to solve computational problems not design problems, etc.

In the era that software development is caught up by time to market, design patterns are the good help. Design pattern can relieve needless effort in trying to invent the proven existing solution as design patterns provides tested, proven development paradigms. Getting along with design pattern is also useful and easy to understand for successors because it has already been well-known in developer society.

As mentioned earlier that there are 3 groups of design patterns classified in [2]. Creational Patterns is the group that we focus. It contains 5 patterns inside, Factory Method, Builder, Abstract Factory, Singleton and Prototype. To improve regular patterns to be able to use with PCS, we pick Builder pattern as it is a foundation and not too complicated.

## V. Builder Pattern

Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations [10]. The general class diagram of Builder pattern is shown in Fig. 1 [10].

Product is a class that has many required methods needed by all ConcreteBuilders for creating objects. Builder is an interface for creating specific product from portions of Product object. ConcreteBuilder constructs and assembles parts of the product by implementing the Builder interface. It defines and keeps track of the specific product it creates and provides method for retrieving that specific one. Director gets order from user and builds specific products from the cooperation of Builder and ConcreteBuilder.

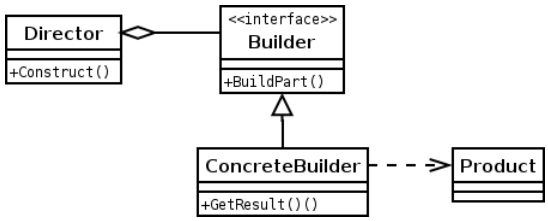


Fig. 1 General Class Diagram of Builder pattern

To be clearer, please see Fig. 2 that shows class diagram of sample Builder pattern and Fig. 3 which is a sequence diagram (with activations) of this case respectively.

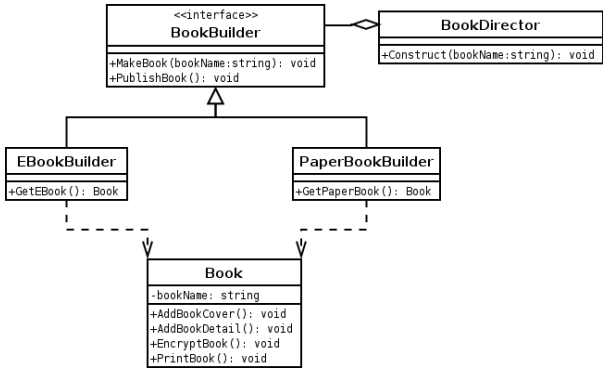


Fig. 2 Class Diagram of sample Builder pattern

The explanations of Fig. 2 are as follows

- Book (is like Product in Fig. 1) is a class containing all required fields and methods those are needed to be chosen, the whole or portions, for creating specific product. Field, in this case, is "private string bookName". There are 4 methods in Book those are "public void AddBookCover()", "public void AddBookDetail()", "public void EncryptBook()" and "public void PrintBook()".
- BookBuilder (is like Builder in Fig. 1) is an interface that has methods for assembling specific products. In Fig. 2, BookBuilder contains "void MakeBook(string bookName)" and "void PublishBook()" abstract methods.
- EBookBuilder and PaperBookBuilder (are like ConcreteBuilder in Fig. 1) are classes those implement BookBuilder interface' methods for building specific products, eBook and paperBook objects. They also have methods for returning specific products, those are just created, which are "public Book GetEBook()" and "public Book GetPaperBook()", respectively.
- BookDirector (is like Director in Fig. 1) gets order from user to build specific products, eBook and paperBook objects, from the cooperation of BookBuilder and EBookBuilder or PaperBookBuilder.

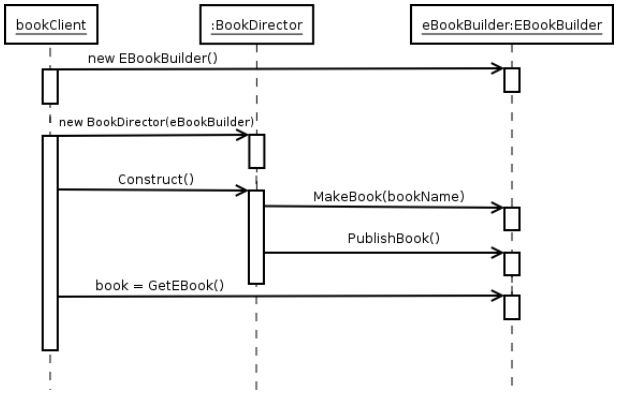


Fig. 3 Sequence Diagram (with Activations) of sample Builder pattern

From Fig. 3 which is a sequence diagram for creating eBook, details of activities are like these

- "bookClient" is any object for starting all activities. It creates object named "eBookBuilder" from EBookBuilder.
- "bookClient" creates object of BookDirector, named "bookDirector", by giving "eBookBuilder" as a parameter. This object will control creational processes of eBook.
- "bookClient" orders "bookDirector" to control building processes by calling "Construct()" method of "bookDirector".
- "bookDirector" instructs "eBookBuilder" (with cooperation of "bookBuilder") to assemble eBook by using "MakeBook(bookName)" and "PublishBook()" methods.
- "bookClient" can get the completed eBook by calling "GetEBook()" method of "eBookBuilder".

To create paperBook, processes are similar to eBook's. We can notice that all essential methods for building any kinds of book are resided in Book class.

## VI. Energy Conscious Builder Pattern Seeking

As regular Builder pattern was not designed intentionally for energy limited situation, so, to be able to work well with such environment, we have to adapt it based on our prior research about C# software writing strategies for PCS [8].

In our experiment, we have studied several techniques to improve original code to use least power, but, we choose to propose just some those are successful or are interested by developers like these

- Builder Pattern that uses only struct instead of class and static methods.

- Builder Pattern that uses Director[1] class, ConcreteBuilder struct and Product class with static Product methods.
- Builder Pattern that uses Director struct, ConcreteBuilder class and Product struct with non-static Product methods.

The diversity of power consumption level is from various combination of struct / class and static / non-static method as they need different level of energy [8].

For more information about power consumption of class and struct, please see Appendix A.

### *A. Builder Pattern That Uses Only Struct and Static Methods.*

Fig. 4 illustrates class diagram of Builder pattern that use struct instead of class and static methods. We try this because most developers think struct is lighter than class while static method consumes less power than non-static one. By the way, these assumptions are not absolutely true since struct is value type and static method consumes a little bit more memory than non-static in some cases.

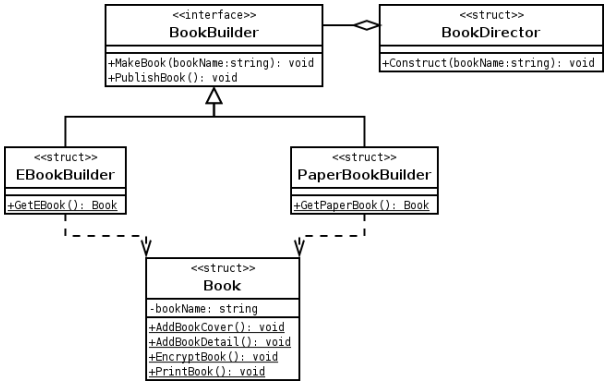


Fig. 4 Class Diagram of Builder pattern that uses only struct and static methods

### *B. Builder Pattern That Uses Director Class, ConcreteBuilder Struct and Product Class with Static Product Methods*

We apply both struct and class while use static method in Product class. Class diagram of this case is shown in Fig. 5.

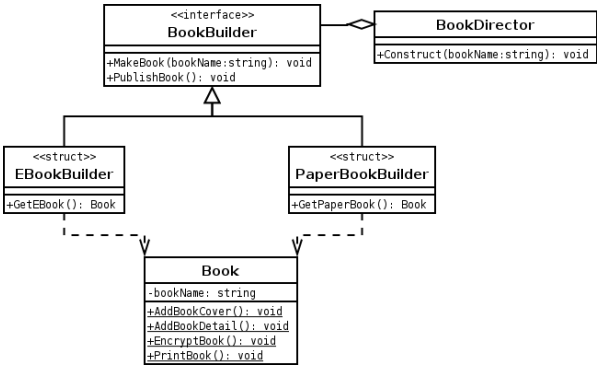


Fig. 5 Class Diagram of Builder pattern that uses Director Class, ConcreteBuilder struct and Product class with static Product methods

[1] Director, ConcreteBuilder and Product are referred to class diagram in Fig. 1 which is the standard diagram for representing Builder Pattern.

### *C. Builder Pattern That Uses Director Struct, ConcreteBuilder Class and Product Struct with Non-Static Product Methods*

This case, we apply struct to some portions. Its class diagram is in Fig. 6.

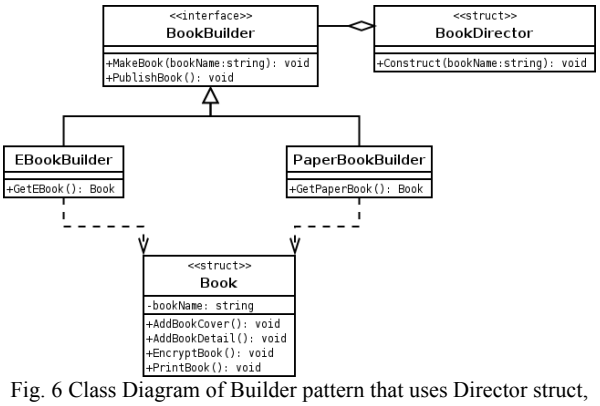


Fig. 6 Class Diagram of Builder pattern that uses Director struct, ConcreteBuilder class and Product struct with Non-static Product methods

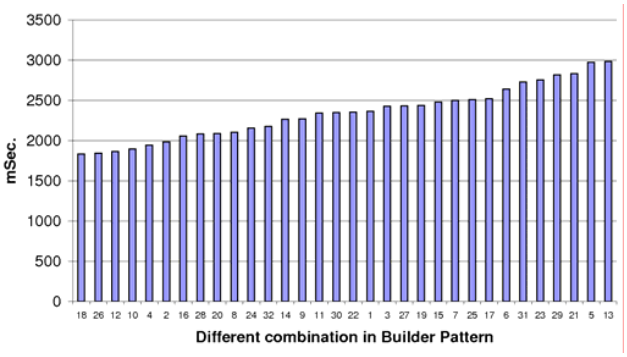


Fig. 7 Different combination consumes unequal CPU consumption

From our experiments, we tested different component combination in Builder pattern and got result of unequal CPU consumption as shown in Fig. 7. We do not mention other combination in detail in this paper due to space limitation. So, the results mentioned in this article are the top best and top worst group in term of energy consumption.

## VII. Measurement and Research Environment

Tiwari mentioned in his paper that time the processor used was directly related to the power it needed [3]. Therefore, to get the same output from similar essential working steps while controlling other kinds of element, the shorter the processor time uses the better performance of the chosen component is - in term of the power optimization.

About the tool in this research, we developed the software, TOM - Time Operation Measurement, which circular checked (every 10 Milliseconds) the timespan the specified process used. TOM will terminate checking itself when the watched process ends. The software can snapshot User Processor Time (UPT), Privileged Processor Time (PPT), Total Processor Time (TPT) and Memory used by the specific software process. UPT is the timespan processor uses just for that

process, PPT is the time processor spends for the operating system to support that process and the TPT is the summation of UPT and PPT.

Besides our own tool, the additional tools from CPU maker and some commercial software are also used. Most of them are the profiler for CPU and memory. CodeAnalyst from AMD is another major tool for checking detail tasks related to CPU. .NET Memory Profiler is another tool for checking memory usage.

This experiment uses both TOM and other profilers since, usually, the profilers are good to see detail information but our research needs both detail and overall result, so, combination of both can provide better completed result.

The results from the measurement shown in this paper were done on the system that used AMD Athlon™ XP 1800+ CPU with 512 MB RAM. The software in the system were regular Microsoft Windows XP SP2, Microsoft .NET Framework Redistributable Package 2.0, the codes to be measured and TOM.

From CPU specification [11], AMD Athlon™ XP 1800+ needs current at 37.7 Ampere with voltage 1.75 Volt. About the memory, from the datasheet [12], its voltage is 2.5 Volt while it needs power 2.9 Watt.

Energy consumed by CPU and memory is calculated by using the same formula

$W = PT$

W = energy (Joule) P = power (Watt) T = time (Second)

However, for memory, information given by manufacturer causes us to use additional formula for calculating power.

$P = IV$

I=current (Ampere) V=voltage (Volt)

## VIII. C# and Intermediate Language (IL) Output of ECBuilder Pattern

Under .NET platform, as Fig. 8 depicts, codes written in C# will be compiled by C# compiler to IL. When user wants to run this program and if there is no native code available, its IL will be compiled again, at run time, by JIT compiler which provides executable native code.

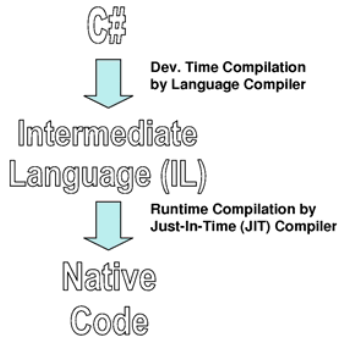


Fig. 8 Two Steps of .NET Compilation

### A. C# Code of Builder Pattern that Consumes Least CPU

Using pure class or struct has some limitations. Though using struct instance takes only 1 step while using class instance takes 2 (as shown in Fig.A1), since struct instance is a value type, when it is used it is needed to be copied and if this copying process is too often, sometimes, using struct can cost in term of power more than class. With this reason, we mix together.

Along with class diagram in Fig.6, List 1 is a code of Builder pattern written in C#. With this combination, Book struct with non-static method, EBookBuilder and PaperBookBuilder class with non-static GetEBook() and non-static GetPaperBook(), and BookDirector struct, our test demonstrated that it consumes least CPU time.

```
// Book struct -  non-static methods in Book
// EBookBuilder|PaperBookBuilder class
// non-static GetEBook | GetPaperBook method
// BookDirector struct
// NOTE: We wrote double d = 2.0;    d = d * 5.0; and
// in methods just for creating some loads for them.
struct Book {
  private string bookName;
  public Book(string bookName) {
    this.bookName = bookName;
    double d = 2.0;    d = d * 5.0;
  }
  public  void AddBookCover() {
    double d = 2.0;    d = d * 5.0;
  }
  public  void AddBookDetail() {
    double d = 2.0;    d = d * 5.0;
  }
  public  void EncryptBook() {
    double d = 2.0;    d = d * 5.0;
  }
  public  void PrintBook() {
    double d = 2.0;    d = d * 5.0;
  }
}
interface BookBuilder {
  void MakeBook(string bookName);
  void PublishBook();
}
class EBookBuilder : BookBuilder {
  private  Book eBook;
  public void MakeBook(string bookName) {
    eBook = new Book(bookName);
    eBook.AddBookDetail();
    eBook.AddBookCover();
  }
  public void PublishBook() {
    eBook.EncryptBook();
  }
  public  Book GetEBook() {
    double d = 2.0;    d = d * 5.0;
    return eBook;
  }
}
class PaperBookBuilder : BookBuilder {
  private  Book paperBook;
  public void MakeBook(string bookName) {
    paperBook = new Book(bookName);
    paperBook.AddBookDetail();
    paperBook.AddBookCover();
  }
  public void PublishBook() {
    paperBook.PrintBook();
  }
  public  Book GetPaperBook() {
    double d = 2.0;    d = d * 5.0;
```

```
    return paperBook;
  }
}
struct BookDirector {
  private BookBuilder bookBuilder;
  public BookDirector(BookBuilder bookBuilder) {
    this.bookBuilder = bookBuilder;
  }
  public void Construct(string bookName) {
    bookBuilder.MakeBook(bookName);
    bookBuilder.PublishBook();
  }
}
public class TestPlainBuilder {
  static void Main(string[] args) {
    BookDirector bookDirector;
    for(double d = 0.0; d < 100000.0; d+=0.01) {
      // Create E-Book
      EBookBuilder eBookBuilder = new EBookBuilder();
      bookDirector = new BookDirector(eBookBuilder);
      bookDirector.Construct("Introduction to C#");
      Book b1 = eBookBuilder.GetEBook();
      // Create Paper Book
      PaperBookBuilder paperBookBuilder = new PaperBookBuilder();
      bookDirector = new BookDirector(paperBookBuilder);
      bookDirector.Construct("Introduction to ASP.NET with C#");
      Book b2 = paperBookBuilder.GetPaperBook();
    }
  }
}
```

List 1 Sample Code in C# of EC Builder Pattern

### *B. Partial IL from Builder Pattern*

Usually IL of class is larger than struct, if everything inside is the same, since class has larger overhead. If there is no constructor, class will have build-in default constructor with 7 byte size while struct will not. General constructor of struct in List 2 has 31 byte size that is smaller when compared with 39 byte general constructor of class in List 3.

List 2 is partial IL of Book struct with Non-static AddBookCover() method. In case of Book is class with static AddBookCover() method, its IL is as shown in List 3 instead.

List 4 shows IL for creating instance of Book struct and calling 2 non-static methods while List 5 is an IL for creating instance of Book class and calling 2 static methods (AddBookCover() and AddBookDetail()).

By the way, using pure struct does not guarantee that the code will consume less energy as the reason from situation mentioned in last part.

```
.class private sequential ansi sealed beforefieldinit Book
       extends [mscorlib]System.ValueType
{
  .field private string bookName
  .method public hidebysig specialname rtspecialname
          instance void  .ctor(string bookName) cil managed
  {
    // Code size       31 (0x1f)
    .maxstack  2
    .locals init (float64 V_0)
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldarg.1
    IL_0003:  stfld      string Book::bookName
    IL_0008:  ldc.r8     2.
    IL_0011:  stloc.0
    IL_0012:  ldloc.0
    IL_0013:  ldc.r8     5.
    IL_001c:  mul
    IL_001d:  stloc.0
    IL_001e:  ret
  } // end of method Book::.ctor
  .method public hidebysig instance void AddBookCover() cil
managed
  {
    // Code size       24 (0x18)
    .maxstack  2
    .locals init (float64 V_0)
    IL_0000:  nop
    IL_0001:  ldc.r8     2.
    IL_000a:  stloc.0
    IL_000b:  ldloc.0
    IL_000c:  ldc.r8     5.
    IL_0015:  mul
    IL_0016:  stloc.0
    IL_0017:  ret
  } // end of method Book::AddBookCover
  ...
}
```

List 2 Partial IL of Book struct and Non-static AddBookCover()

```
.class private auto ansi beforefieldinit Book
       extends [mscorlib]System.Object
{
  .field private string bookName
  .method public hidebysig specialname rtspecialname
          instance void  .ctor(string bookName) cil managed
  {
    // Code size       39 (0x27)
    .maxstack  2
    .locals init (float64 V_0)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  nop
    IL_0008:  ldarg.0
    IL_0009:  ldarg.1
    IL_000a:  stfld      string Book::bookName
    IL_000f:  ldc.r8     2.
    IL_0018:  stloc.0
    IL_0019:  ldloc.0
    IL_001a:  ldc.r8     5.
    IL_0023:  mul
    IL_0024:  stloc.0
    IL_0025:  nop
    IL_0026:  ret
  } // end of method Book::.ctor
  .method public hidebysig static void  AddBookCover() cil managed
  {
    // Code size       24 (0x18)
    .maxstack  2
    .locals init (float64 V_0)
    IL_0000:  nop
    IL_0001:  ldc.r8     2.
    IL_000a:  stloc.0
    IL_000b:  ldloc.0
    IL_000c:  ldc.r8     5.
    IL_0015:  mul
    IL_0016:  stloc.0
    IL_0017:  ret
  } // end of method Book::AddBookCover
  ...
}
```

List 3 Partial IL of Book class and static AddBookCover()

```
.method public hidebysig newslot virtual final
        instance void  MakeBook(string bookName) cil managed
{
  // Code size       38 (0x26)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldarg.0
```

```
    IL_0002:  ldarg.1
    IL_0003:  newobj     instance void Book::.ctor(string)
    IL_0008:  stfld      valuetype Book EBookBuilder::eBook
    IL_000d:  ldarg.0
    IL_000e:  ldflda     valuetype Book EBookBuilder::eBook
    IL_0013:  call       instance void Book::AddBookDetail()
    IL_0018:  nop
    IL_0019:  ldarg.0
    IL_001a:  ldflda     valuetype Book EBookBuilder::eBook
    IL_001f:  call       instance void Book::AddBookCover()
    IL_0024:  nop
    IL_0025:  ret
  } // end of method EBookBuilder::MakeBook
```

List 4 Partial IL of creating instance of Book struct and calling non-static method

```
  .method public hidebysig newslot virtual final
          instance void  MakeBook(string bookName) cil managed
  {
    // Code size       26 (0x1a)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldarg.1
    IL_0003:  newobj     instance void Book::.ctor(string)
    IL_0008:  stfld      class Book EBookBuilder::eBook
    IL_000d:  call       void Book::AddBookDetail()
    IL_0012:  nop
    IL_0013:  call       void Book::AddBookCover()
    IL_0018:  nop
    IL_0019:  ret
  } // end of method EBookBuilder::MakeBook
```

List 5 Partial IL of creating instance of Book class and calling static method

## IX. Results of Experiments

TOM and AMD profiler give result in the same way. Results of CPU and RAM consumption from various combinations of Builder pattern's components measured by TOM are shown in Table 2. Fig.9 is the result in chart format.

PC-nPM-CBC-nCBM-DC (Book is class with Non-static method, EBookBuilder and PaperBookBuilder are class with Non-static methods and BookDirector is class) is regular Builder pattern. As mentioned earlier, it was not designed for power optimization, so, its CPU usage in the table is not low.

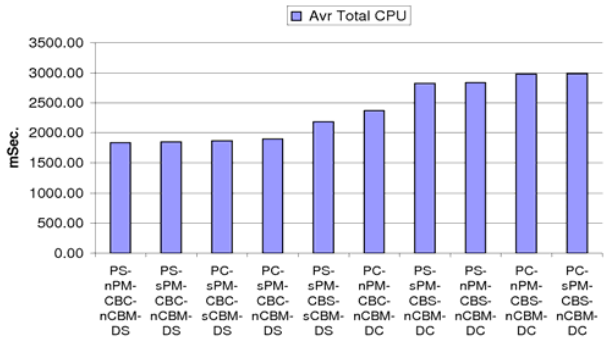


Fig. 9 CPU Usage Comparison

From our prior study [8], it illustrated that struct consumed less CPU than class as it was lighter. It is right in regular manner. However, as struct is value type, if its size is large and it has to be passed around often, it may consume more CPU than class which is reference type.

AMD CodeAnalyst shows that ConcreteBuilders (EBookBuilder & PaperBookBuilder) are portions those consume much CPU while other parts in Builder pattern do not require much CPU. One reason is ConcreteBuilder objects are often used and passed around. With this reason, for them, being reference type is better than value type. On the other hand, Director (BookDirector) is not passed much, so, being struct is appropriated as it is lighter.

The result from TOM shown in Table II and Fig. 9 also supports these issues as the best 4 builders use ConcreteBuilder class and Director struct while the worst 4 ones use ConcreteBuilder struct and Director class.

From Table II, the group of good combination consumes less CPU than regular Builder pattern around 20% while the group of bad combination consumes more CPU than regular one also around 20%.

The result from Table II shows that using only struct and static methods are not the best choice for Builder pattern as it decreases CPU consumption only around 8%.

Memory usages from all experiments are not much different. The range is around less than 3%.

UPT *User Processor Time*
PPT *Privilege Processor Time*
TPT *Total Processor Time*
PC *Product Class*
PS *Product Struct*
nPM *Non-Static Product Method*
sPM *Static Product Method*
CBC *Concrete Builder Class*
CBS *Concrete Builder Struct*
nCBM *Non-Static Concrete Builder Method*
sCBM *Static Concrete Builder Method*
DC *Director Class*
DS *Director Struct*

TABLE II
CPU TIME AND MEMORY USAGE FROM VARIOUS KIND OF BUILDER

| CODE COMPONENT | UPT (mSec.) | PPT (mSec.) | TPT (mSec.) | RAM (kBytes) | CPU Diff.(%) | RAM Diff.(%) |
|---|---|---|---|---|---|---|
| **PS-nPM-CBC-nCBM-DS** | **1801.59** | **31.71** | **1833.30** | **4068.80** | **-22.51** | **1.13** |
| PS-sPM-CBC-nCBM-DS | 1814.61 | 31.71 | 1846.32 | 4089.47 | -21.96 | 1.64 |
| PC-sPM-CBC-sCBM-DS | 1826.29 | 37.72 | 1864.01 | 4052.80 | -21.21 | 0.73 |
| PC-sPM-CBC-nCBM-DS | 1860.01 | 38.39 | 1898.40 | 4085.47 | -19.75 | 1.54 |
| PS-sPM-CBS-sCBM-DS | 2152.43 | 27.71 | 2180.13 | 4113.33 | -7.85 | 2.24 |
| ***PC-nPM-CBC-nCBM-DC*** | ***2317.00*** | ***48.74*** | ***2365.74*** | ***4023.33*** | ***0.00*** | ***0.00*** |
| PS-sPM-CBS-nCBM-DC | 2785.00 | 35.05 | 2820.06 | 4155.20 | 19.20 | 3.28 |
| PS-nPM-CBS-nCBM-DC | 2795.35 | 39.39 | 2834.74 | 4136.67 | 19.83 | 2.82 |
| PC-nPM-CBS-nCBM-DC | 2927.21 | 49.07 | 2976.28 | 4108.27 | 25.81 | 2.11 |
| PC-sPM-CBS-nCBM-DC | 2943.23 | 41.06 | 2984.29 | 4145.60 | 26.15 | 3.04 |

Energy usages from both CPU and Ram of various Builder patterns are demonstrated in Table III while Fig. 10 compares them.

TABLE III
ENERGY USAGE FROM VARIOUS KIND OF BUILDER

| CODE CHARACTERISTIC | CPU (Joule) | Mem (Joule) | Total (Joule) | Diff % |
|---|---|---|---|---|
| **PS-nPM-CBC-nCBM-DS** | 120.96 | 0.04 | 121.00 | -22.51 |
| PS-sPM-CBC-nCBM-DS | 121.82 | 0.04 | 121.86 | -21.96 |
| PC-sPM-CBC-sCBM-DS | 122.99 | 0.04 | 123.03 | -21.21 |
| PC-sPM-CBC-nCBM-DS | 125.26 | 0.04 | 125.30 | -19.75 |
| PS-sPM-CBS-sCBM-DS | 143.85 | 0.05 | 143.89 | -7.84 |
| ***PC-nPM-CBC-nCBM-DC*** | 156.09 | 0.05 | 156.14 | 0.00 |
| PS-sPM-CBS-nCBM-DC | 186.07 | 0.06 | 186.13 | 19.21 |
| PS-nPM-CBS-nCBM-DC | 187.04 | 0.06 | 187.10 | 19.83 |
| PC-nPM-CBS-nCBM-DC | 196.37 | 0.07 | 196.44 | 25.81 |
| PC-sPM-CBS-nCBM-DC | 196.90 | 0.07 | 196.97 | 26.15 |

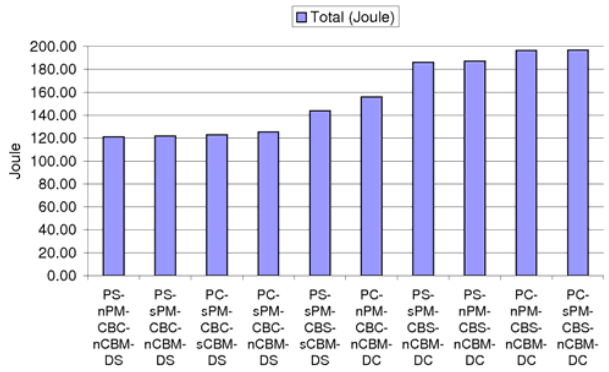

Fig. 10 Energy Usage Comparison

## X. CONCLUSION

Design Patterns are conglomeration of proven solutions for repeated common object oriented programming problems. Builder Pattern is one of them. Though all patterns have been accepted by OOP developers as a great technique, they were not originally designed for power optimization which is an essential requirement for mobile devices. This research focuses in improving Builder Pattern to be appropriated for power limitation environment. Finally, we found the solution, Energy Conscious Builder Pattern, which is combination of Director stuct, ConcreteBuilder class. Our ECBuilder Pattern consumes less energy than regular Builder Pattern around 20%.

## XI. FUTURE WORKS

There are other kinds of Design Patterns those can be optimized to use less energy to be better used with battery powered devices. Our research team is going to do these.

## APPENDIX A

Usually, class consumes more power than struct does. One of the reasons is, as shown in Fig.A1, invoking class instance needs 2 steps of operation while just one is required by struct. Another reason is class instance is stored in heap which has Garbage Collector (GC) that surely increases CPU load while. Though the result of our prior experiment [8] mentioned that struct consumes less CPU time than class, we have to be careful when using struct with large sized data. Because struct is a value type, to pass it around, the system needs to copy its whole content every moving time. If copying process of large sized data is too often, energy needed for this process can be significant and, sometimes, can be more than load of GC. Microsoft recommended the struct size should be less than 16 bytes [13].

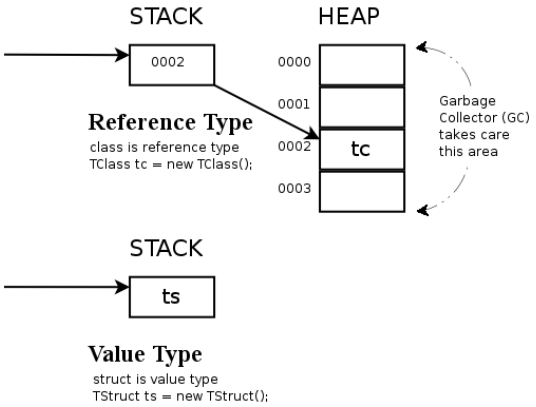

Fig. A1 Invoking instance of Class and Struct

## REFERENCES

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides The Gang of four [Online]. Available: http://hillside.net

[2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Massachusetts: Addison-Wesley, 1995.
[3] V. Tiwari, S. Malik, and A.Wolfe. Power analysis of embedded software: A first step towards software power minimization. In IEEE Transaction VLSI Systems, December 1994.
[4] V. Tiwari, S. Malik and A. Wolfe. Compilation Techniques for Low Energy: An Overview. In 1994 Symposium on Low-Power Electronics, San Diego, CA, October 1994.
[5] Y-T S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, December 1997.
[6] Peymandoust, T. Simunic and G.D. Micheli. Low Power Embedded Software Optimization using Symbolic Algebra. In IEEE Proceeding of the 2002 Design, Automation and Test in Europe Conference and Exhibition, 2002.
[7] Chatzigeorgiou and G. Stephanides. Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors. In Ada-Europe 2002, 2002.
[8] (Journal Online Sources style) K. Chantarasathaporn and C. Srisa-an. (2006, January). Object-Oriented Programming Strategies in C# for Power Conscious System. *International Journal of Computer Science* [Online]. Volume 1 Number 1. Available: http://www.enformatika.org/ijcs/v1/v1-1-7.pdf
[9] http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29
[10] http://www.dofactory.com/Patterns/PatternBuilder.aspx
[11] http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24309.pdf
[12] http://www.valueram.com/datasheets/KVR266X64SC25_512.pdf
[13] Kumar, S. Struct in C#. Available: http://www.codeproject.com/csharp/structs_in_csharp.asp