多言語パラダイムを前提とした設計手法

～計算モデルとプログラミング言語の有効活用へ～

マイクロソフト株式会社
デベロッパー & プラットフォーム統括本部
エバンジェリスト
荒井　省三

アジェンダ

- 分析・設計技術
- 分析・設計に必要とされる要素技術
  - プログラミング言語のマルチパラダイム化
- ダックタイプ論
  - オブジェクト コンテナー
  - オブジェクト 構成コンテナー
  - 検証コンテナー
- 関数型パラダイム論
  - タスク間のギャップ
  - 非同期プログラミング モデル
  - 非同期ワークフロー
  - 非同期プログラミングの課題
- 多言語パラダイムで設計するために
- まとめ

分析・設計技術

メンタル モデル (パターンの結末)

水の流れる方法 (自然)

Alexander パターン

Software パターン

トヨタ ウェイ

アジャイル

スクラム

リーン アーキテククチャ

- アレグザンダー
  - 自然に潜む秩序の性質を捉える
  - パターン言語へ
- ダイナブック構想
  - 人の支援
  - メンタルモデル
- オブジェクト指向の欠点
  - メンタルモデルの欠如
  - プログラミング言語はクラスが中心
- メンタルモデルを重視
  - アジャイル
  - 適切なコンテキスト

From Patterns: Eastward to Lean, Westward to true Object–James O Coplien より引用

ユビキタス言語 (モデル表現)

ユビキタス言語

設計の技術的側面

技術用語

技術の設計パターン

ドメイン モデル
の用語

境界付き
コンテキスト名

大規模構造の
専門用語

パターン集

開発者は
ビジネス用語に
対して無知

ユーザーが使う
ビジネス用語は
設計に表出しない

Domain Driven Design–Eric Evans より引用

関心の分離 (SoC)

Keep model
Unified by

CONTINUOUS
INTEGRATION

BOUND
CONTEXT

names
enter

Assess/overview
Relathionship with

SHARD
KERNEL

CONTEXT
MAP

UBIQUITOUS
LANGUAGE

…

…

…

…

…

境界コンテキストで
コア ドメインと
サブドメインに分離する

Domain Driven Design–Eric Evans より引用

2 種類のオブジェクトを分離

関心	中心となる操作	DCI アーキテクチャ
ユーザーのゴール	ドメイン オブジェクトに対する直接操作	ゴールを達成するためのタスク フロー
要求	ステートマシン、 正規化	ユースケース
技術	従来のオブジェクト指向	マルチパラダイム設計 DCI
設計の焦点	データの形式	アルゴリズムの形式
スコープ	1 つのオブジェクトか 静的に決まる相互作用	動的な関連付けを持つ 複数のオブジェクト
相互作用	名詞 ー 動詞	動詞 ー 名詞
例	文字の削除 残高の出力	スペルチェック 送金

※ DCI: Data Context Interaction

From Patterns: Eastward to Lean, Westward to true Object–James O Coplien より引用

振る舞いはコンテキストへ

From Patterns: Eastward to Lean, Westward to true Object–James O Coplien より引用

振る舞い (ロール) のマッピング

Methodful Roles

a, b, c

d, e, f

g, h, k

クラス定義

d, e, f
m, n, p

g, h, k
q, r, s

d, e, f
t, u, v
g, h, k

Methodless Role (Identifires)

g, h, k

d, e, f

a, b, c

オブジェクト

http://www.artima.com/articles/dci_visionP.html より引用

振る舞い (ロール) のマッピング

Methodful Roles

a, b, c

d, e, f

g, h, k

クラス定義

d, e, f

m, n, p

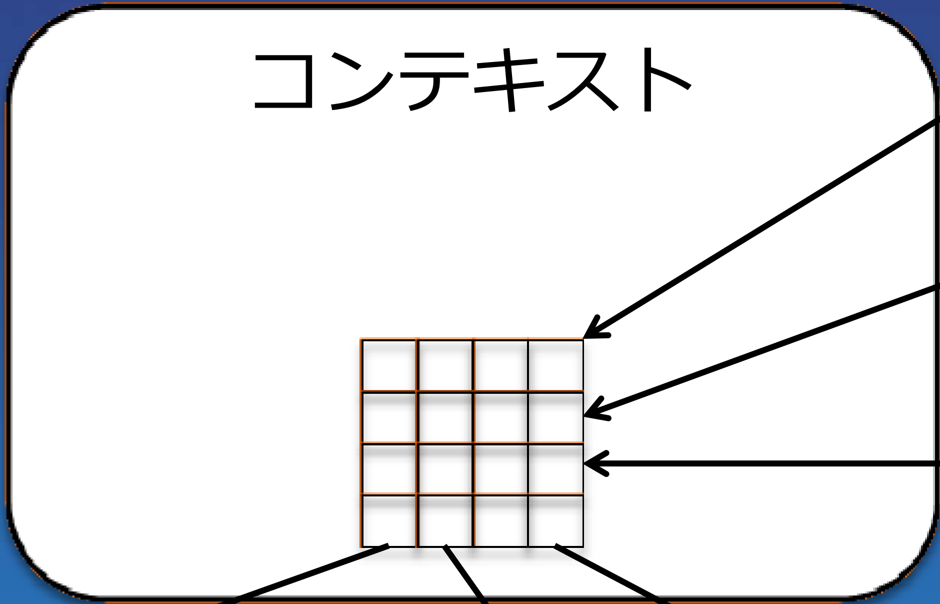
g, h, k

q, r, s

d, e, f

t, u, v

g, h, k



Methodless Role (Identifires)

g, h, k

d, e, f

a, b, c

オブジェクト

http://www.artima.com/articles/dci_visionP.html より引用

ユースケースを使った関心の分離

<<use case slice>>

<<non use case slice>>

<<non use case slice>>

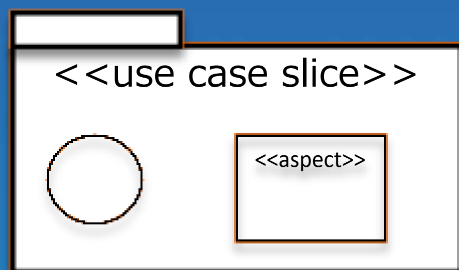
<<extend>>

<<use case slice>>

<<non use case slice>>

ユースケーススライス

ユースケース独立スライス
(アスペクト – 非同期、保守性、 etc -)



アプリケーション層やドメイン層などの設計構造へマップする

Command Query Responsibility Segregation

ユーザーインターフェース

クエリー

ビューデータ

イベント

Background Worker

※Eventually Consistency

Background Worker

コマンド

データストア

分析・設計に必要とされる要素技術

計算モデルの必要性

- 部分障害、分散、並列、非同期、一時的な非一貫性を前提とした、クラウドやマルチ コア全体の振る舞いを抽象的にモデル化する基盤が必要 = 計算モデル
  - 意味論の明確化と検証 = アーキテクチャの確立
- 歴史は古いが、クラウドやマルチ コアで再注目
  - 分散・並列への対応が必要 = 実行コンテキスト
- ネットワーク遅延問題は解決できていない
  - システムのボトルネックは、普遍
CPU > キャッシュ > メモリ > HDD > 回線

計算モデルとプログラミング言語

計算モデル

オートマン
  チューリング マシン
    ノイマン型 (RAM)

機能的関数モデル
ラムダ計算モデル
自然演繹モデル

並列
  DAG
  共有メモリ モデル
  ネットワーク モデル

  CSP
  アクター モデル

パラダイム

手続型 (命令型)

宣言型
  論理型
  関数型

オブジェクト指向

動的型 / 静的型

言語

C / C++ / Basic
FORTRAN /
COBOL

Prolog

Haskell / Scala /
Ocaml / Erlang /
F#

Lisp

Java / C# /
Smalltalk
Ruby / Python

プログラミング モデルの方向性

Parallelism
(並列)

Distributed
(分散)

Reactive
(反応)

Concurrent
(同時実行)

 ※ リアクティブ プログラミングの代表例は、イベント ドリブン

プログラミング言語の分類

オブジェクト指向パラダイム

Smalltalk

OCaml

Java C# Scala F#
C++ Ruby Python JavaScript

C PASCAL Haskell

Prolog

LISP Scheme Clojure

手続型言語 関数型言語 論理型言語

命令型言語 宣言型言語

http://www. artonx.org/diary/20100604.html より引用

プログラミング言語の存在意義

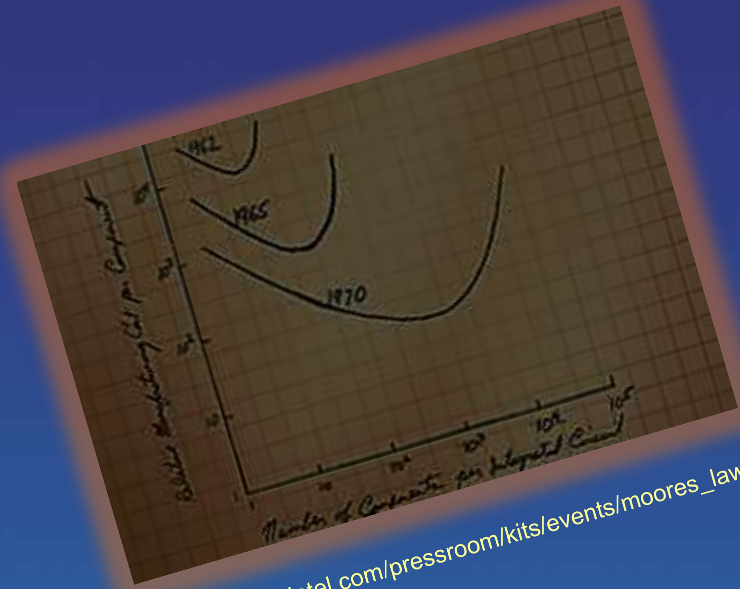
特定の問題領域 ⇔ 対応する言語

- 科学技術計算
  - FORTRAN
- 事務計算
  - COBOL

- 数学や論理学
  - 関数型や論理型言語
- 汎用目的
  - C 、Java 、 C# 、etc

現代の時代背景

- ムーアの法則
  - メニーコア シフト
  - NUMA の台頭
  - …



http://www.intel.com/pressroom/kits/events/moores_law_40th/

- できなかった事への挑戦
  - ラムダ式のエッセンス
  - クロージャー
  - 集合演算
  - …

```
λ x . x * 2

[] |> List. map (fun x-> …)
```

善か悪か-万能論-

- 汎用言語 (GPL) の万能論
  - 是 か 否 か
- メジャーな言語が収束したか?
  - 多種多様な言語生態系
  - 言語同士が切磋琢磨
- 理想 と 現実
  - 答えはあるのか?

多言語パラダイム = 現実路線

- 適材適所でプログラミング言語を組合わせる
  - 目的に応じた使い分け
- 現実的か？
  - 学習する言語を増やしたくない
  - メンテナーが不在になる
  - 新しいことを覚えたくない
- トレードオフ
  - 必要とする機能と工数の駆け引き

ダックタイプ論

具体例の提案

- **DI コンテナー**
- **構成 コンテナー**
- **検証 コンテナー**

DI コンテナーを提案する理由

- 静的プログラミング言語で、 DI コンテナーが良く使われている
- 動的言語で DI コンテナが使われていない
  - オブジェクト間の結合度が緩い
  - 変更が容易
  - 静的プログラミング言語の問題点が問題にならない

DI コンテナーを使用するメリット

- オブジェクト間の結合度を緩くする
  - 契約による設計
- オブジェクトの入れ替えを容易にする
  - モック オブジェクト
  - テスト容易性
- 依存性の注入
  - ゲッター、セッターに対する注入
  - アスペクトの注入

DI コンテナーを使用するハードル

- コンテナーの作成にかかる労力
  - 汎用的に利用できるものは工数がかかる
- 公開されているソフトウェアを使う
  - Spring.NET や Seaser .NET など
- 少しだけ使いたい時のハードルが高い
  - 汎用的であればあるほど
  - 管理するライブラリを減らしたい
  - …

オブジェクトを疎結合にする

- 名前や引数という契約だけにする
  - dynamic キーワードで実現できる
- メリット
  - 動的にオブジェクトを入れ替えられる (リリース後にビルドしない)
  - オブジェクト生成の戦略をスクリプト内にカプセル化
    - スクリプトクラス、初期化、メソッド アスペクト、etc
  - 少しの工数でコンテナーを作成できる

動的コンテナー

```
public dynamic GetObject(string objectName)
{
  var code =
      GetSourceFromFile(objectName.To_Ruby()) ;
  var source =
_engine.CreateScriptSourceFromString(code) ;
//オブジェクトを戻すスクリプトを実行する
  dynamic obj = source.Execute(_scope) ;

  return obj;
}
```

動的コンテナー

利用者

GetObject

Object

コンテナー

スクリプトクラス(モック)

スクリプトクラス(委譲)

スクリプトファクトリー

オブジェクト生成の戦略をスクリプトにカプセル化

オブジェクト生成の戦略

モックオブジェジェクト：スクリプトで用意可能

委譲：スクリプトで継承クラスを実装

ファクトリー：最適なインスタンスを返す実装

動的コンテナーのデメリット

- メンバー名解決のオーバーヘッド
  - 名前によるメンバー ポインターの取得
  - メンバーの呼び出し

  - 静的言語は、コンパイル時にメンバー ポインターが解決される
- インテリセンスが使用できない
  - 実行時の名前解決のため
  - 条件コンパイルなど方法は考えられる

ダックタイプ論

具体例の提案

- **DI コンテナー**
- **構成 コンテナー**
- **検証 コンテナー**

オブジェクトの設定への応用

- オブジェクト インスタンスに対する初期設定など
- 初期化ルールは変更になる可能性が高い

- インスタンス生成までは今迄のパラダイム
  - 動的コンテナーよりも早い

- 構成コンテナーを設計する
  - 構成規約のみを設計 (呼出し方法)
  - 構成ロジックをスクリプトで実現する

構成コンテナー

利用者

Configure

Object

コンテナー

構成スクリプト

ダックタイプ論

具体例の提案

DI コンテナー

構成 コンテナー

検証 コンテナー

検証ルールへの応用

- 検証ルールは、業務要望によって変化する可能性が大きい
- 将来の可能性を考慮した設計でも、対応できない場合がある
  - 柔軟性のある規則の実現には工数がかかる
  - テスト パターンも増大していく
- 検証コンテナーを設計する
  - 使い方の規則のみを設計 (呼出し方法、検証結果)
  - 検証ロジックをスクリプトで実現する

検証コンテナー

利用者

Validate

Object

コンテナー

検証スクリプト

まとめ

- ダックタイプを適切に理解する
  - 柔軟な戦略を実現するパラダイム
  - 採用にはトレードオフがある

  - 柔軟な戦略を低コストで実現が可能

- 利用する場 (コンテキスト) が重要
  - 実行コンテキスト
  - 保守コンテキスト
  - 将来的な変化
    - 予見できないことへの戦略を練る

関数型パラダイム論

- 非同期の活用
  - 非同期プログラミング
  - アクター モデル

関数型パラダイムとは

- 関数を中心とした考え方
- Google の Map Reduce フレームワークは map 関数と reduce 関数を使用している
  - 大規模な分散コンピューティング環境
  - Hadoop のような OSS 実装もある

- 論理学や数学の課題を解決するために、手続型パラダイムとは異なるアプローチ

- 現代の時代背景
  - メニーコア
  - 大量なメインメモリ
  - ・・・
- 増大したコンピューター リソースを効率良く、活用する手法が必要
  - マルチスレッド、スレッド プール、同時実行ランタイムなど

タスク間のギャップを埋めるには

- インメモリ リソース
  - 高速動作が可能
- 外部リソース
  - アクセスする経路によって様々な性能
- リソース アクセス速度
  - キャッシュ > メインメモリ > HDD > ネットワーク など
- 解決のために非同期パターンが用いられる

非同期プログラミングという課題

- 共通言語ランタイムの非同期プログラミング モデル (APM) のサポート
  - BeginXXXX/Endxxxx パターン (IAsyncResult を利用)
  - XXXXAsync/xxxxCompleted パターン (イベントを利用)
- 容易に構築できることがメリット

Begin/End パターン

```
//非同期のファイル IO
open System.IO
let openFile fileName =
//ファイルストリームを開きます
  use fs = new FileStream(fileName, FileMode.Open)
//ファイルの長さのバイト配列を作成します
  let data = Array.create(int fs.Length) 0uy
//コールバックされる関数を定義します
  let callback ar =
      fs.EndRead(ar) |> ignore
//ハンドルを解放します
      ar.AsyncWaitHandle.Close()
//ファイルから読み込みます
  let asyncResult =
      fs.BeginRead(data, 0, data. Length, (fun ar-> callback ar) ,
null)
//コールバックが終了するまで待ちます
  asyncResult.AsyncWaitHandle.WaitOne() |> ignore
  data
```

非同期パターンの複雑さ

主スレッド

Begin → End

Begin → End

Begin → End

- より複雑化していく
- スレッドの同期化が難しい

非同期ワークフロー

```
open System.IO
let openFile fileName =
  async {
//ファイルストリームを開きます
    use fs = new FileStream(fileName,
FileMode.Open)
//ファイルの長さのバイト配列を作成します
    let data = Array.create(int fs.Length) 0uy
//ファイルから読み込みます
    let! asyncResult =
        fs.AsyncRead(data, 0, data. Length)
//データを戻します
    return data
}
```

リアクティブ

非同期ワークフローのメリット

- 非同期パターンを組合わせた複雑度を下げる
- コードは手続型のように見える
  - 保守が容易
  - 記述も容易

- 実行方法の選択肢
  - 同期的実行
  - メッセージ エージェント
  - 非同期実行 (スレッド プール)
  - 非同期実行 (タスク並列)

非同期処理の課題

- ユーザー インターフェースとの連携
  - 同期的に使用しない
    - UI スレッドのデッド ロック問題
  - リアクティブ プログラミング モデルを利用
    - コールバック チャネルを使ったメッセージ
    - イベント通知
  - ポーリング
- バックグラウンド処理
  - リソース競合を検討する
  - フォアグラウンドとの連携パターンを検討する

関数型パラダイム論

非同期への活用
- 非同期プログラミング
- アクター モデル

メッセージ エージェント

スレッド

エージェント

処理

結果

キュー

メイン

Msg

ポスト

受信

リプライ要求

リプライ チャネル

リアクティブ パターン

イベントの活用

リアクティブ パターンへ

マネージャー

ダウンローダー

ダウンローダー操作

イベントの購読
イベントの発行

UI スレッド

非同期
ワークフロー

イベント発生

実行スレッド

同期コンテキスト

まとめ

- 関数型パラダイムの特徴を理解する
- 並列コンピューティングへの応用
  - 非同期ワークフロー
  - アクター モデル
  - リアクティブ プログラミング
- 利用する場 (コンテキスト) が重要
  - 実行コンテキスト
並列化、分散コンピューティング (クラウドなど)
  - 処理量が増大するコンテキスト

Visual Studio 非同期プログラミング
– 将来的な拡張 –

非同期プログラミングモデルの拡張

- .NET Framework 4 でタスク（Task）モデルを追加
- 非同期プログラミング モデルを拡張
  - APM（Asynchronous Programming Model) - Begin/End パターン
  - EAP（Event based Asynchronous Programming model） - APM をラップして提供
  - TAP（Task based Asynchronous Programming model） - タスクを使った新しい非同期プログラミング モデル
- Visual Studio Async CTP
  - .NET Framework 4
  - Silverlight 4
  - Windows Phone 7

サンプル

```
public async void AsyncIntroSingle()
{
    WriteLinePageTitle(
        await new WebClient().DownloadStringTaskAsync(
                             new Uri("http://www.weather.gov")));
}

// 今までのパターンでは
public void AsyncIntroSingleBefore()
{   var client = new WebClient();
    client.DownloadStringCompleted +=
              AsyncIntroSingleBefore_DownloadStringCompleted;
    client.DownloadStringAsync(new Uri("http://www.weather.gov"));
}
void AsyncIntroSingleBefore_DownloadStringCompleted(object sender,
DownloadStringCompletedEventArgs e)
{    WriteLinePageTitle(e.Result);
}
```

サンプル

```
public async void AsyncFromAPM()
{
    var response =
        await WebRequest.Create("http://www.weather.gov")
                        .GetResponseAsync();
    var stream = response.GetResponseStream();
    var buffer = new byte[1024];
    int count;
    while ((count =
             await ReadAsync(stream, buffer, 0, 1024)) > 0)
    {
        Console.Write(
             Encoding.UTF8.GetString(buffer, 0, count));
    }
}
```

サンプル

```
public static Task<int> ReadAsync(
      Stream stream, byte[] buffer, int offset, int count)
{
    var tcs = new TaskCompletionSource<int>();
    stream.BeginRead(buffer, offset, count, iar =>
    {
        try { tcs.TrySetResult(stream.EndRead(iar)); }
        catch (Exception exc) { tcs.TrySetException(exc); }
    }, null);
    return tcs.Task;
}
```

Async CTP の特徴

- Async キーワード
  - メソッド定義に使用する
  - 非同期の開始と終了操作を制御するためのコードが追加される
- Await キーワード
  - メソッド呼び出しに使用する
  - 非同期操作の完了に対する完了待ちコードが追加される
- キーワードを使って、コンパイラが必要なコードを生成する = キーワードは DSL の一種
- 拡張メソッドをライブラリとして提供
- 非同期操作を逐次操作のように記述できる

まとめ

- 分析・設計手法
  - コンテキストによるドメイン分割
  - DCI アーキテクチャ
  - CQRS アーキテクチャ スタイル
  - コンテキストを適切に抽出して、制約に応じた技術を柔軟に導入していく
- 多言語パラダイムとは、適材適所でプログラミング言語を組合わせる
  - 言語の特徴を理解
  - 適用できるパラダイムを分析する

リファレンス

Lean Software Architechture
http://www.leansoftwarearchitecture.com/

The DCI Architecture: A New Vision of Object-Oriented Programming
http://www.artima.com/articles/dci_visionP.htmle

Domain Driven Design Community
http://domaindrivendesign.org/

ユースケースによるアスペクト指向ソフトウェア開発
http://www.seshop.com/product/detail/6798/

Context for Goal-level Product Line Derivation
http://www.disi.unitn.it/~pgiorgio/publications.html

リファレンス

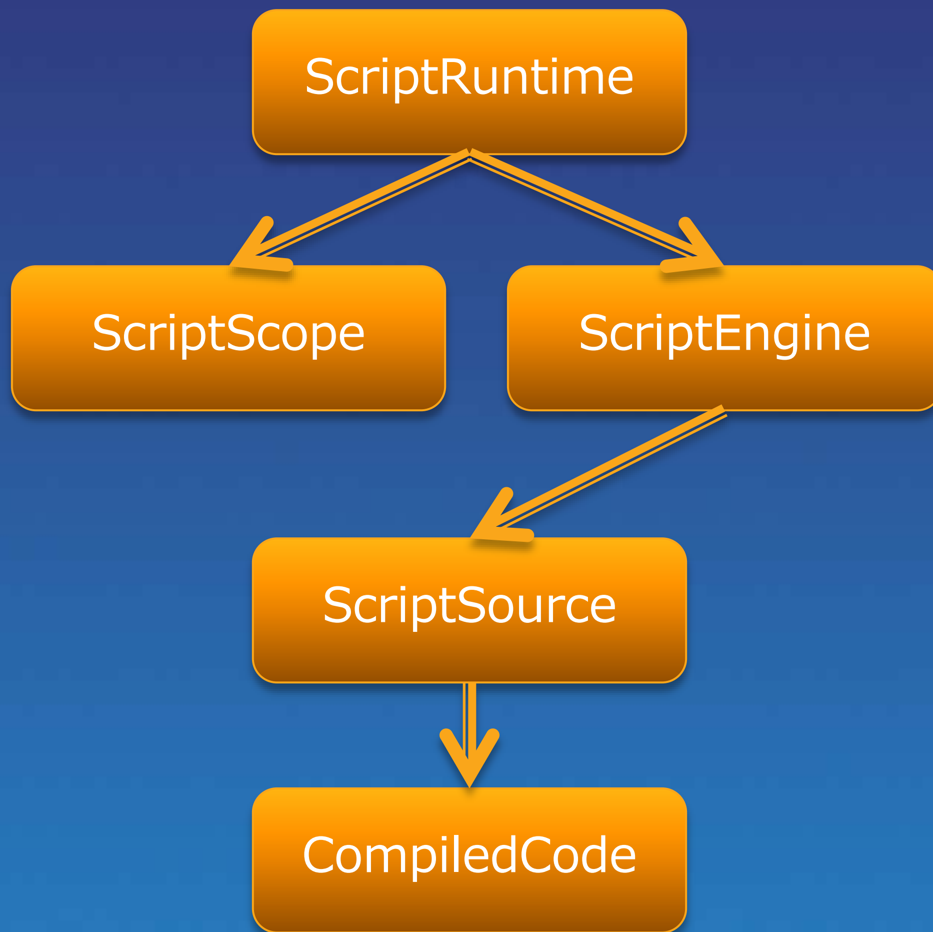
Workshop on Modeling and Reasoning in Context シリーズ
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/

The Journal of Object Technology
http://www.jot.fm/index.html

ご清聴ありがとうございました。

Appendix

DLR ホスティング (既存の言語)



DLR ホスティング (独自言語)

ConsoleHost

Provider

LanguageContext

Run()

CompileSourceCode()

ScriptCode

非同期ワークフローの応用 1

```
// Begin ・ End パターンを応用するために
type System.IO.Stream with
  [<CompiledName("AsyncRead")>]
  member stream.AsyncRead(buffer: byte[],?offset,?count) =
    let offset = defaultArg offset 0
    let count  = defaultArg count buffer.Length
    // Async.FromBeginEnd メソッドがポイント
    Async.FromBeginEnd (buffer,offset,count,
                         stream.BeginRead,stream.EndRead)

    // 以下のように書き換えることができる
    Async.FromBeginEnd (
           (fun cb, s ->
                 stream.BeginRead(buffer, offset, count, cb, s)),
            stream.EndRead)
```

非同期ワークフローの応用 2 (1/3)

```
// Async ・ Completed イベント パターンを応用するために
type System.Net.WebClient with
  [<CompiledName("AsyncDownloadString")>]
  member this.AsyncDownloadString (address:Uri) : Async<string> =
    // 継続用の非同期ワークフローを作成します (処理、例外、キャンセル)
    let downloadAsync =
      // FromContinuations メソッド
      // ( ('T -> unit) * (exn -> unit) *
      // (OperationCanceledException -> unit)
      //    -> unit)      -> Async<'T>
      // 'T：型、exn：例外、OperationCancel：キャンセル)
      Async.FromContinuations (fun (cont, econt, ccont) ->
        let userToken = new obj()
        // イベント ハンドラーを定義
        let rec handler =
```

非同期ワークフローの応用 2 (2/3)

```
// デリゲートの定義を行います
System.Net.DownloadStringCompletedEventHandler
 ( fun _ args ->
  if userToken = args.UserState then
    // イベントが発生したらハンドラーを削除
    this.DownloadStringCompleted.RemoveHandler(handler)
    if args.Cancelled then
      // キャンセル時の処理
      ccont (new OperationCanceledException())
    elif args.Error <> null then
      // エラー時の処理
      econt args.Error
    else
      // 結果を返す
      cont args.Result
)
```

非同期ワークフローの応用 2 (3/3)

```
    // イベント ハンドラーを登録します
    this.DownloadStringCompleted.AddHandler(handler)
    // Async 呼出しを行います
    this.DownloadStringAsync(address, userToken)
)

// キャンセル時の処理を使用する、非同期ワークフローを返します
async {
  use! _holder = Async.OnCancel(fun _ -> this.CancelAsync())
  return! downloadAsync
}
```

キャンセル可能なエージェント

```
let agent (token:CancellationToken) =
  MailboxProcessor.Start(
    (fun inbox ->
      let loop () = async {
        let results = new ResizeArray<_>()
        while not token.IsCancellationRequested do
          // メッセージを受け取る
          let! ms = inbox.Receive()
          match ms with
          | Cont(url,name) ->
               do Console.WriteLine("MSG=" + name)
               do results.Add(name)
          | Fetch replyChannel ->
               replyChannel.Reply(results)
        }
      // 処理を開始します
      loop() )
  , cancellationToken = token)
```

宣言的なイベント処理

```
// win は、 Window インスタンス
let location = win.MouseDown
  // マウスがクリックされたイベントをフィルタリングする
  //  info は、 MouseEventArgs
  |> Event.filter (fun info ->
                     info.Button = MouseButtons.Right)
  // イベント引数を座標 x,y に変換します
  |> Event.map (fun info ->
                  sprintf "{%d, %d}" info.X info.Y)
// 処理を行うイベント ハンドラーを登録します
//    msg は、 map 関数で作成した座標
Location.Add(fun msg -> label1.Text <- msg)
```

F# では event モジュールだけでなく Observable モジュールも用意して、リアクティブ プログラミングを支援する