## Nie DI do DI

1. *Tutaj wymagania spisane w pracy*
2. Obszar zainteresowan: Program, EventRepository, CalendarEvent.
3. Klasa Program jest nieelastyczna, dodawanie if’ow
4. EventRepository podobnie – mozliwe przeladowanie getEvents, addEvent
5. Zaczynamy od Logiki biznesowej – srodkowej warstwy aplikacji trojwarstwowej. Wymaganie dot. *Spotkania* tj Meeting
   1. Wydzielenie wspolnego dla klas interfejsu IcalendarEvent
   2. Implementacja w klasie Meeting + dolozenie *Participants+tostring* za pomoca meetingToString
   3. Uwspólnienie w klasie bazowej, przenieść zawartość **CalendarEvent** (ktore bedzie ToDo) do klasy bazowej CalendarEventBase
   4. Dodac bool CanShareTime {get;} do CalendarEventBase i get set do bazowej
   5. Atrybuty serializacji we wszystkich klasach
6. Program sie buduje, ale dodanie spotkania do *Program* = kolejny if, ale
   1. Niemozliwy rownolegly development,
   2. Kod jest brzydki,nietestowalny itd  
      Wymarzona funkcja main= minimalna ilosc kodu
7. Folder UI i klase OptionDispatcher – to co jest w main’nie teraz, w dwoch slowach: ChooseOptionAndRun = ChooseOption + Run
8. Przeniesc kod z Program do ChooseOption-odpowiedzialnosc:**wybrac opcje**

Rozdzial pomiedzy tym jak sie opcja przedstawia i jaki klawisz jest jej przypisany, ale to ciagle jest opcja – uwspolnic pod jednym typem. Zmienic tablice stringow na IEnumerable<IOption>. Nastepnie dodac implementacje :

private readonly IEnumerable<IOption> options;

public OptionsDispatcher(IEnumerable<IOption> options)

{

this.options = options;

}

public bool ChooseOptionAndRun()

{

IOption option = PrintAndChooseOption();

bool result = option.Run();

return result;

}

private IOption PrintAndChooseOption()

{

PrintAvailableOptions();

return ChooseOption();

}

private IOption ChooseOption()

{

while (true)

{

string chosenOptionAsString = Console.ReadLine();

IOption chosenOption = options.FirstOrDefault(o => o.MatchesString(chosenOptionAsString));

if (chosenOption != default(IOption))

{

return chosenOption;

}

Console.WriteLine("Incorrect option, try again");

}

}

private void PrintAvailableOptions()

{

Console.WriteLine("Pick option:");

foreach (var option in options)

{

Console.WriteLine(option);

}

}

Nie zmuszajmy klasy do tworzenia zaleznosci – jej odpowiedzialnoscia jest **wybrac opcje** nie stworzyc opcje. Zaleta – stan jest niezmienny. **Zmiana** listy **opcji** **nie powoduje zmiany kodu OptionsDispatcher’a.**

Implementacja opcji AddTodo

internal const string AddToDoOptionString = "a";

private readonly IEventsRepository eventsRepository;

public AddTodoOption(IEventsRepository eventsRepository)

{

this.eventsRepository = eventsRepository;

}

public bool MatchesString(string chosenOptionAsString)

{

bool result = StringComparer.InvariantCultureIgnoreCase.Equals(chosenOptionAsString, AddToDoOptionString);

return result;

}

public sealed override string ToString()

{

return AddToDoOptionString + " - todo";

}

public bool Run()

{

DateSpan schedule = DateSpanReader.PromptForDateSpan();

Console.Write("Title: ");

string title = Console.ReadLine();

Todo calendarEvent = new Todo(schedule, title);

eventsRepository.AddEvent(calendarEvent);

return true;

}

Jesli AddTodoOption byloby tworzone w OptionsDispatcher’rze, to OptionsDispatcher bylby odpowiedzialny za tworzenie zależności (EventRepository) dla wszystkich swoich opcji. Jest duzo opcji, zadna nie jest dobra, nie ma dobrego miejsca w dispatcherze, do stworzenia zaleznosci. EventRepository jest globalne, moze byc singleton’em, bo jest wspoldzielone, ale to jest kiepskie.

Brakuje opcji do wypisywania zdarzen. Przy DI nie ma dylematy, gdzie obiekt bedzie tworzony i jak bedzie przekazywany. Lista zdarzeń:

internal class ListEventsOption : IOption

{

internal const string ListEventsOptionString = "l";

private readonly IEventsRepository eventsRepository;

public ListEventsOption(IEventsRepository eventsRepository)

{

this.eventsRepository = eventsRepository;

}

public bool MatchesString(string chosenOptionAsString)

{

return StringComparer.InvariantCultureIgnoreCase.Equals(ListEventsOptionString, chosenOptionAsString);

}

public bool Run()

{

ICalendarEvent[] calendarEvents = eventsRepository.GetEvents(DateSpan.Max);

foreach (var calendarEvent in calendarEvents)

{

Console.WriteLine(calendarEvent);

}

return true;

}

public sealed override string ToString()

{

return ListEventsOptionString + " - list events";

}

}

AddMeetingOption

1. Skopiowac pobieranie daty i tytulu z AddTodoOption
2. **promptForParticipants** snippet
3. Zwrocic uwage, jak latwo dodac kolejna opcje do OptionsDispatcher’a  
     
    class AddMeetingOption : IOption

{

private const string MeetingOptionAsString = "m";

private readonly IEventsRepository eventsRepository;

public AddMeetingOption(IEventsRepository eventsRepository)

{

this.eventsRepository = eventsRepository;

}

public bool MatchesString(string chosenOptionAsString)

{

return StringComparer.InvariantCultureIgnoreCase.Equals(MeetingOptionAsString, chosenOptionAsString);

}

public bool Run()

{

DateSpan schedule = DateSpanReader.PromptForDateSpan();

Console.Write("Title: ");

string title = Console.ReadLine();

string[] participants = PromptForParticipants();

Meeting meetingToAdd = new Meeting(schedule, title, participants);

eventsRepository.AddEvent(meetingToAdd);

return true;

}

private string[] PromptForParticipants() // snippet!!

{

Console.Write("Participants' names separated with comas: ");

string participantsAsString = Console.ReadLine();

return participantsAsString != null

? participantsAsString.Split(new[] { ',' }, StringSplitOptions.RemoveEmptyEntries)

: new string[0];

}

public sealed override string ToString()

{

return MeetingOptionAsString + " - meeting";

}

}

internal class EndApplicationOption : IOption // snippet!!

{

private const string ApplicationEndsString = "e";

public bool MatchesString(string chosenOptionAsString)

{

return StringComparer.InvariantCulture.Equals(chosenOptionAsString, ApplicationEndsString);

}

public bool Run()

{

return false;

}

public override string ToString()

{

return ApplicationEndsString + " - application terminates";

}

}

EventsRepository eventsRepository = new EventsRepository();

IEnumerable<IOption> options = new IOption[]

{

new AddTodoOption(eventsRepository),

new ListEventsOption(eventsRepository),

new AddMeetingOption(eventsRepository),

new EndApplicationOption()

};

OptionsDispatcher optionsDispatcher = new OptionsDispatcher(options);

bool continueRunning = true;

while (continueRunning)

{

continueRunning = optionsDispatcher.ChooseOptionAndRun();

}

Zostalo wymaganie dotyczace dodawania spotkan TYLKO jesli nie ma innych spotkan, w tym czasie. TODO mozna dodawac zawsze. Potrzebne cos, co opakuje kawalek logiki decydujacej o tym, czy dodac spotkanie do listy, czy nie. Nazwijmy to polityka dodawania.

1. Na poczatek sam interfejs IaddPolicy, bedzie mial 2 implementacje, dla dwoch przypadkow, albo doda do repozytorium, albo nie

Napisac testy dla planner’a:  
  
 [TestFixture]

class PlannerTests

{

[Test]

public void ShouldUseExclusiveSchedulePolicyWhenEventCannotShareTime()

{

ICalendarEvent calendarEvent = Substitute.For<ICalendarEvent>();

calendarEvent.CanShareTime.Returns(false);

IAddPolicy exclusiveScheduleAddPolicy = Substitute.For<IAddPolicy>();

IAddPolicy shareableTimeAddPolicy = Substitute.For<IAddPolicy>();

Planner planner = new Planner(exclusiveScheduleAddPolicy, shareableTimeAddPolicy);

planner.AddEvent(calendarEvent);

exclusiveScheduleAddPolicy.Received(1).TryAddToRepository(calendarEvent);

}

[Test]

public void ShoulduseShareableSchedulePolicyWhenEventCanShareTime()

{

ICalendarEvent calendarEvent = Substitute.For<ICalendarEvent>();

calendarEvent.CanShareTime.Returns(true);

IAddPolicy exclusiveScheduleAddPolicy = Substitute.For<IAddPolicy>();

IAddPolicy shareableTimeAddPolicy = Substitute.For<IAddPolicy>();

Planner planner = new Planner(exclusiveScheduleAddPolicy, shareableTimeAddPolicy);

planner.AddEvent(calendarEvent);

shareableTimeAddPolicy.Received(1).TryAddToRepository(calendarEvent);

}

}

I samego planner’a:

internal class Planner

{

private readonly IAddPolicy \_exclusiveScheduleAddPolicy;

private readonly IAddPolicy \_shareableTimeAddPolicy;

public Planner(IAddPolicy exclusiveScheduleAddPolicy, IAddPolicy shareableTimeAddPolicy)

{

\_exclusiveScheduleAddPolicy = exclusiveScheduleAddPolicy;

\_shareableTimeAddPolicy = shareableTimeAddPolicy;

}

public void AddEvent(ICalendarEvent calendarEvent)

{

if (calendarEvent.CanShareTime)

\_shareableTimeAddPolicy.TryAddToRepository(calendarEvent);

else

\_exclusiveScheduleAddPolicy.TryAddToRepository(calendarEvent);

}

}

1. Latwo jest pisac testy i mock’i, jesli uzywamy DI – latwo sie podmienia zaleznosci
2. Nie trzeba miec zaimplementowanych funkcjonalnosc, zatem mozna implementowac ‘od gory do dolu  
     
    internal class ExclusiveSchedulePolicy : IAddPolicy

{

private readonly IEventsRepository eventsRepository;

public ExclusiveSchedulePolicy(IEventsRepository eventsRepository)…

public void TryAddToRepository(ICalendarEvent eventToAdd)

{

ICalendarEvent[] intersectingEvents = eventsRepository.GetEvents(eventToAdd.Schedule);

if (intersectingEvents.All(ie => ie.CanShareTime))

{

eventsRepository.AddEvent(eventToAdd);

}

}

}

class ShareableTimePolicy : IAddPolicy

{

private readonly IEventsRepository eventsRepository;

public ShareableTimePolicy(IEventsRepository eventsRepository)

{

this.eventsRepository = eventsRepository;

}

public void TryAddToRepository(ICalendarEvent calendarEvent)

{

eventsRepository.AddEvent(calendarEvent);

}

}

1. Wydzielic interfejs z Planner’a
2. Zmodyfikowac AddTodoOption i AddMeetingOption zeby uzywaly planner’a  
     
     
   ShareableTimePolicy shareableTimeAddPolicy =

new ShareableTimePolicy(eventsRepository);

ExclusiveSchedulePolicy exclusiveTimeAddPolicy =

new ExclusiveSchedulePolicy (eventsRepository);

IPlanner planner =

new Planner(exclusiveTimeAddPolicy, shareableTimeAddPolicy);

/\* UWAGA NA KOLEJNOSC W KONSTRUKTORZE PLANNERA\*/

IEnumerable<IOption> options = new IOption[]

{

new AddTodoOption(planner),

new ListEventsOption(eventsRepository),

new AddMeetingOption(planner),

new EndApplicationOption()

};

Wyprobowac aplikacje – w tym miejscu NIE DZIALA, nie ma ustawiania CanShareTime.

Potrzeba dodac fabryki dla Todo i Meeting, ktore beda konfigurowalne z poziomu Program’u. Tworzenie obiektów w AddMeetingOption i AddTodoOption nie spełnia oczekiwań – przykład tworzenia obiektów ‘na sztywno’.

public interface ITodoFactory

{

Todo Create(DateSpan schedule, string title);

}

internal interface IMeetingFactory

{

Meeting Create(DateSpan schedule, string title, string[] participants);

}

Dodanie abstrakcji nad tworzeniem meeting i todo do opcji, zwieksza mozliwosc ich przetestowania. To co uniemozliwia napisanie unit testow, to referencja do Console. Podobnie w OptionsDispatcher.

Dodac do option dispatcher TextReader – wstrzykiwanie zaleznosci infrastrukturalnych.

class TodoFactory : ITodoFactory

{

private bool canShareTime;

public TodoFactory(bool canShareTime)

{

this.canShareTime = canShareTime;

}

public Todo Create(DateSpan schedule, string title)

{

return new Todo(schedule, title) { CanShareTime = canShareTime };

}

}

Wiele osób uważa za wygodne również konfigurowanie aplikacji w korzeniu kompozycji, możemy dorzucic nazwę pliku z danymi

EventsRepository eventsRepository = new EventsRepository("calendarData.dat");

Z wymaganych funkcjonalnosci brakuje jeszcze logowania.

1. Stworzyc folder logging
2. Dodac logger’a ze snippetu logger
3. Wyciagnac interfejs Ilogger
4. Dodac using(Logger logger = new Logger()){} w Program oraz dodac do konstruktorów opcji (AddTodoOption, AddMeetingOption etc.) oraz do OptionDispatcher’a. Opakowac Run i ChooseOption

Cross-cutting concerns – wymagania przekrojowe – w opcjach daloby sie rozwiazac problem metoda szablonowa, ale jesli logowanie dotyczy kilku klas ‘niespokrewnionych’ ze soba, wtedy jest to problem...