Declarative Design Pattern Development using Aspect Oriented Programming

Mario Luca Bernardi\textsuperscript{1}  
Marta Cimitile\textsuperscript{2}  
Giuseppe A. Di Lucca\textsuperscript{1}

\textsuperscript{1}University of Sannio, Benevento, Italy  
\textsuperscript{2}Te.L.Ma. University, Rome, Italy

SAC 2014, 24-28 March  
Gyeongju, Korea
Motivation behind DPMF

- Design patterns (DPs) provide well-known solutions to recurring problems that developers have to face during software implementation.
- Aspect Oriented Software development (AOSD) can be used to improve the way software is structured, decomposed and implemented reducing crosscutting concerns.
- Model-driven Software Development can be used to perform (semi-)automatic generation of both design and code starting from high level models that hide the complexity and make the consequences of trade-offs less binding.

but

- DPs may impact modularity and flexibility introducing crosscutting concerns and make the overall system too much centered on initial design choices.
- AOSD requires the knowledge of aspect-oriented languages and the re-design of existing design patterns and implementation using an aspect language.
- MDSD alone needs complex tooling to be effective and often suffers of round-trip engineering and models-artifacts synchronization problems.
Motivation behind DPMF

- Design patterns (DPs) provide well-known solutions to recurring problems that developers have to face during software implementation.
- Aspect Oriented Software development (AOSD) can be used to improve the way software is structured, decomposed and implemented reducing crosscutting concerns.
- Model-driven Software Development can be used to perform (semi-)automatic generation of both design and code starting from high level models that hide the complexity and make the consequences of trade-offs less binding.

but

- DPs may impact modularity and flexibility introducing crosscutting concerns and make the overall system too much centered on initial design choices.
- AOSD requires the knowledge of aspect-oriented languages and the re-design of existing design patterns and implementation using an aspect language.
- MDSD alone needs complex tooling to be effective and often suffers of round-trip engineering and models-artifacts synchronization problems.
Motivation behind DPMF

- Design patterns (DPs) provide well-known solutions to recurring problems that developers have to face during software implementation.
- Aspect Oriented Software development (AOSD) can be used to improve the way software is structured, decomposed and implemented reducing crosscutting concerns.
- Model-driven Software Development can be used to perform (semi-)automatic generation of both design and code starting from high level models that hide the complexity and make the consequences of trade-offs less binding.

but

- DPs may impact modularity and flexibility introducing crosscutting concerns and make the overall system too much centered on initial design choices.
- AOSD requires the knowledge of aspect-oriented languages and the re-design of existing design patterns and implementation using an aspect language.
- MDSD alone needs complex tooling to be effective and often suffers of round-trip engineering and models-artifacts synchronization problems.
The proposed DPFM integrated approach aims:

- to express and implement DP **declaratively** describing them using a DSL linked to a meta-model
- to use a completely **dynamic approach** in which system classes are **oblivious** of being involved in a pattern collaboration
- to avoid code generation for final system classes: generated code should only **change behaviour dynamically**, at run-time.
The proposed DPFM integrated approach aims to express and implement DP **declaratively** describing them using a DSL linked to a meta-model.

- to use a completely **dynamic approach** in which system classes are **oblivious** of being involved in a pattern collaboration
- to avoid code generation for final system classes: generated code should only **change behaviour dynamically**, at run-time.
The proposed DPFM integrated approach aims

- to express and implement DP **declaratively** describing them using a DSL linked to a meta-model
- to use a completely **dynamic approach** in which system classes are **oblivious** of being involved in a pattern collaboration
- to avoid code generation for final system classes: generated code should only **change behaviour dynamically**, at run-time.
The proposed DPFM integrated approach aims to express and implement DP **declaratively** describing them using a DSL linked to a meta-model. It employs a completely **dynamic approach** in which system classes are **oblivious** of being involved in a pattern collaboration. Also, it avoids code generation for final system classes: generated code should only **change behaviour dynamically**, at run-time.

In short, DPFM can be regarded as a DSL-driven aspect weaver: it can be instructed on which system classes to modify **at run-time** in order to implement a specified design.
The overall structure of the metamodel (and derived DSL) is based on:

- a core language part (that is Java in the current prototype) allowing the representation of code elements
- a DP part introducing the main concepts of the DSL language:
  - concerns
  - composite crosscutting definitions
- The model introduces several crosscutting definitions exposed as DSL statements (e.g. role definition, implementation and assignment, composition and injection rules) that are used to perform dynamic interception of a wide range of events in order to inject desired behaviours.
The overall structure of the metamodel (and derived DSL) is based on:

- a core language part (that is Java in the current prototype) allowing the representation of code elements
- a DP part introducing the main concepts of the DSL language:
  - concerns
  - composite crosscutting definitions

The model introduces several crosscutting definitions exposed as DSL statements (e.g. role definition, implementation and assignment, composition and injection rules) that are used to perform dynamic interception of a wide range of events in order to inject desired behaviours.
The overall structure of the metamodel (and derived DSL) is based on:

- a core language part (that is Java in the current prototype) allowing the representation of code elements
- a DP part introducing the main concepts of the DSL language:
  - concerns
  - composite crosscutting definitions
- The model introduces several crosscutting definitions exposed as DSL statements (e.g. role definition, implementation and assignment, composition and injection rules) that are used to perform dynamic interception of a wide range of events in order to inject desired behaviours.
DPMF framework is centered on few key goals:

- **transparent**: since its application should not bind the original system to design choices, libraries or languages that makes difficult to remove the framework itself or change original design decisions.

- **expressive**: users should be able to express their design using an high level declarative DSL.

- **dynamic**: generated source code is never statically added to the system source code. DSL statements are used only to change dynamic behaviour of the system when the generated aspects are linked to it.
DPMF framework is centered on few key goals:

- **Transparent**: since its application should not bind the original system to design choices, libraries or languages that makes difficult to remove the framework itself or change original design decisions.

- **Expressive**: users should be able to express their design using an high level declarative DSL.

- **Dynamic**: generated source code is never statically added to the system source code. DSL statements are used only to change dynamic behaviour of the system when the generated aspects are linked to it.
DPMF framework is centered on few key goals:

- **transparent**: since its application should not bind the original system to design choices, libraries or languages that makes difficult to remove the framework itself or change original design decisions.

- **expressive**: users should be able to express their design using an high level declarative DSL.

- **dynamic**: generated source code is never statically added to the system source code. DSL statements are used only to change dynamic behaviour of the system when the generated aspects are linked to it.
The Design Pattern Modeling Framework Architecture

An overview of the DPMF Architecture

- The foundations are based entirely on the Eclipse modeling platform.
- The middle layer implements the DSL parsing strategy taking DP specifications from a repository. The parsed DSL is used by an AspectJ-based support library that is linked to the original system (and modified as needed).
- The top layer is the IDE layer used to interact with the framework users. It provides a DSL editor used to write specifications and some views showing an overview of the pattern structure.
The Design Pattern Modeling Framework Architecture

An overview of the DPMF Architecture

DPMF IDE Architecture

- The foundations are based entirely on the Eclipse modeling platform.
- The middle layer implements the DSL parsing strategy taking DP specifications from a repository. The parsed DSL is used by an AspectJ-based support library that is linked to the original system (and modified as needed).
- The top layer is the IDE layer used to interact with the framework users. It provides a DSL editor used to write specifications and some views showing an overview of the pattern structure.
DPMF IDE Architecture

- The foundations are based entirely on the Eclipse modeling platform.
- The middle layer implements the DSL parsing strategy taking DP specifications from a repository. The parsed DSL is used by an AspectJ-based support library that is linked to the original system (and modified as needed).
- The top layer is the IDE layer used to interact with the framework users. It provides a DSL editor used to write specifications and some views showing an overview of the pattern structure.
The Case Study Setup

A real Java system was designed and developed by two different expert groups:
- one group adopted DPFM;
- the other used a classic OOP design.

The assessment of the DPMF approach takes into account two key aspects:
- the quality of the modularization (evaluated using DOS/DOF metrics)
- the impact on system source code with respect to requirement changes involving adopted patterns (evaluating %cloned-LOC by module and changes to DOS/DOF).

The system, a highly-available back-end serving multimedia items for a web/mobile online store, is written in Java and is comprised of 78684 LOC, 454 classes and 33 interfaces.
The Case Study Setup

- A real Java system was designed and developed by two different expert groups:
  - one group adopted DPFM;
  - the other used a classic OOP design.
- The assessment of the DPFM approach takes into account two key aspects:
  - the quality of the modularization (evaluated using DOS/DOF metrics)
  - the impact on system source code with respect to requirement changes involving adopted patterns (evaluating %cloned-LOC by module and changes to DOS/DOF).
- The system, a highly-available back-end serving multimedia items for a web/mobile online store, is written in Java and is comprised of 78684 LOC, 454 classes and 33 interfaces.
The Case Study Setup

- A real Java system was designed and developed by two different expert groups:
  - one group adopted DPFM;
  - the other used a classic OOP design.
- The assessment of the DPMF approach takes into account two key aspects:
  - the quality of the modularization (evaluated using DOS/DOF metrics)
  - the impact on system source code with respect to requirement changes involving
    adopted patterns (evaluating %cloned-LOC by module and changes to DOS/DOF).
- The system, a highly-available back-end serving multimedia items for a
  web/mobile online store, is written in Java and is comprised of 78684 LOC, 454
  classes and 33 interfaces.
The Case Study Setup

Main components of the designed system

Modularity evaluation with Cloned LOC ratio and DOF
The LOC ratio (%LOC) of each design pattern concern over the cloned LOC ratio are compared for each module in both DPMF (AOP) and OOP versions. The results show that the ratio of the average cloned LOCs in the OOP implementation is significantly greater w.r.t. the DPMF system.

Figure shows a change in requirement had a quite different impact on OOP and DPMF systems. In the OOP case, the number of source code changes (for both the number of affected LOC and the number of affected modules) is much greater than the DPMF case.
Degree of Focus (DOF) of the modules is reported for Items implementing Composite and Command hierarchy. Modules of the OOP version have a worse DOF since they explicitly implement design patterns protocols in addition to other secondary concerns. In DPMF system the DP logic is better modularized in the aspects that are derived by the framework from the DSL statements and super-imposed at run-time.
AOSD and MDSD features have been exploited to develop an approach and a supporting framework allowing

- the declarative specification of DPs by a DSL;
- an oblivious AOP-based implementation that can be detached from the system with reduced impact;
- to improve the modularity and the internal code quality.
AOSD and MDSD features have been exploited to develop an approach and a supporting framework allowing

- the declarative specification of DPs by a DSL;
- an oblivious AOP-based implementation that can be detached from the system with reduced impact;
- to improve the modularity and the internal code quality.
AOSD and MDSD features have been exploited to develop an approach and a supporting framework allowing

- the declarative specification of DPs by a DSL;
- an oblivious AOP-based implementation that can be detached from the system with reduced impact;
- to improve the modularity and the internal code quality.
AOSD and MDSD features have been exploited to develop an approach and a supporting framework allowing

- the declarative specification of DPs by a DSL;
- an oblivious AOP-based implementation that can be detached from the system with reduced impact;
- to improve the modularity and the internal code quality.

The results from a case study showed that the AOP implementation of DPs significantly improved the modularity of the system with respect to traditional OO version.
Future Work

- Improvements of both the prototype framework and DSL;
- A larger case study including a controlled experiment to compare maintenance effort in OOP and DPFM scenarios;
- A richer set of design constructs, in order to express more complex design and architectural patterns.
Future Work

- Improvements of both the prototype framework and DSL;
- A larger case study including a controlled experiment to compare maintenance effort in OOP and DPFM scenarios;
- A richer set of design constructs, in order to express more complex design and architectural patterns.
Future Work

- Improvements of both the prototype framework and DSL;
- A larger case study including a controlled experiment to compare maintenance effort in OOP and DPFM scenarios;
- A richer set of design constructs, in order to express more complex design and architectural patterns.
Thank you for listening!

Any questions?