Models And Aspects

Handling Cross-Cutting Concerns in the context of MDSD

Markus Völter
voelter@acm.org
www.voelter.de

Martin Lippert
lippert@acm.org
www.martinlippert.org

About us

völter.de
ingenieurbüro für softwaretechnologie

Markus Völter

voelter@acm.org
www.voelter.de

- Independent Consultant
- Based out of Heidenheim, Germany
- Focus on
  - Software Architecture
  - Middleware
  - Model-Driven Software Development

it-agile

Martin Lippert

lippert@acm.org
www.martinlippert.org

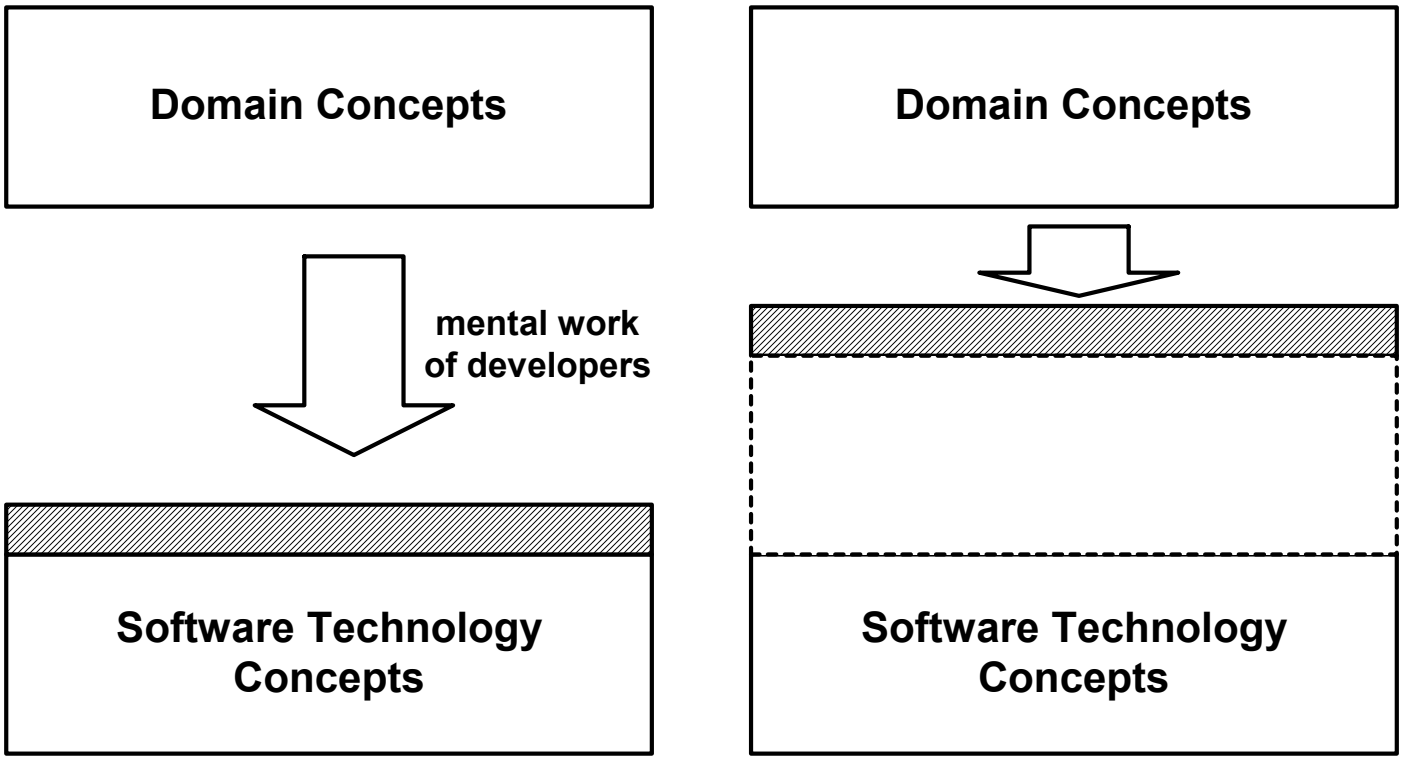
- Consultant at it-agile GmbH
- Hamburg, Germany
- Focus on
  - Software Architecture
  - Agile Software Development
  - Eclipse-Technology

CONTENTS

CONTENTS

What is MDSD?

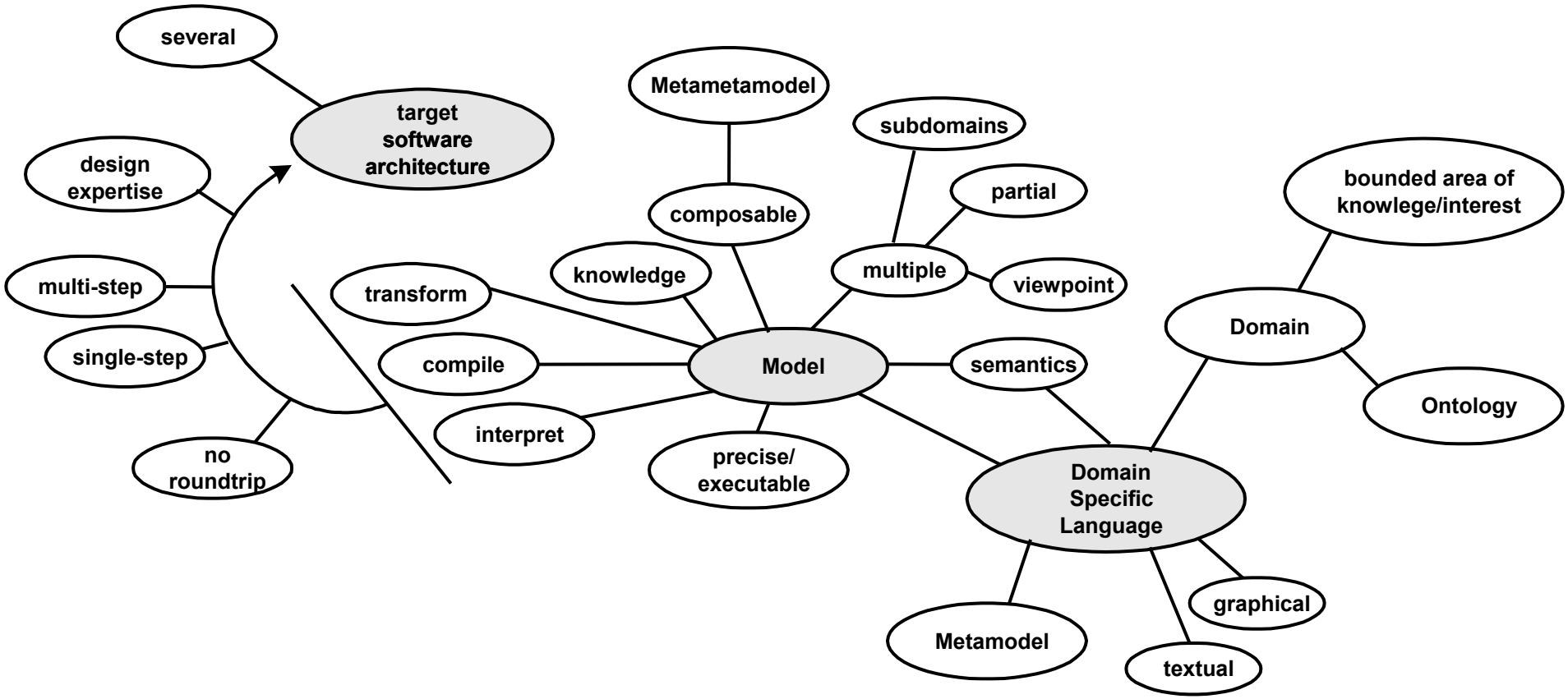
- Domain Driven Development is about making software development more **domain-related** as opposed to **computing related**. It is also about making software development in a certain domain **more efficient**.



What is MDSD? II

- Model-Driven Software Development is about **making models first class development artefacts** as opposed to "just pictures".
- Various aspects of a system are not programmed manually; rather they are **specified using a suitable modeling language**.
- The language for expressing these models is specific to the domain for which the models are relevant. The modeling languages used to describe such models are called **domain-specific languages** (DSL).
- Models have to be **translated into executable code** for a specific platform.
  - Such a translation is implemented using **model transformations.**
  - An approach based on **model interpretation** is also possible, but seldomly used – I will ignore this here!

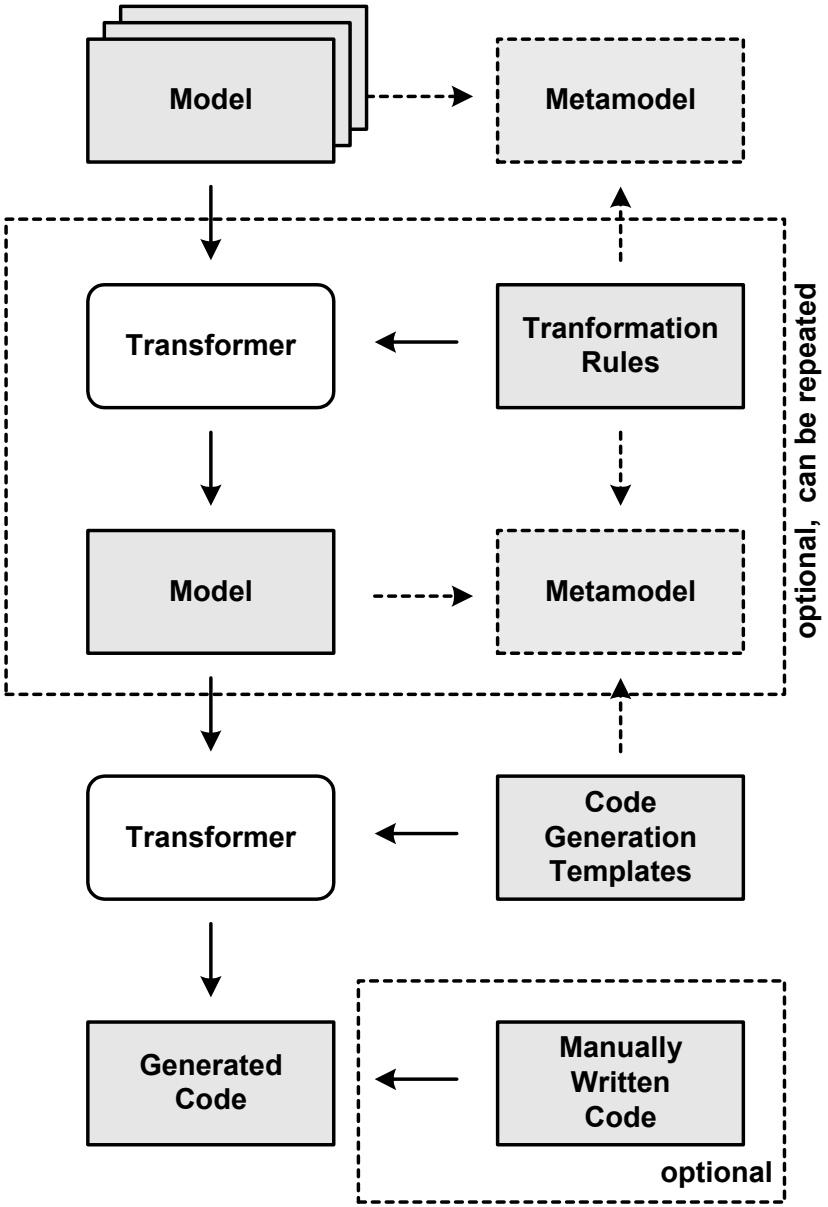
What is MDSD? III



- Related Approaches (Specializations): MDA, SF, DSM, GP, …

How does MDSD work?

- Developer develops **model(s)** based on certain metamodel(s).
- Using **code generation templates**, the model is transformed to executable code.
- Optionally, the **generated code is merged** with manually written code.
- One or more **model-to-model transformation steps** may precede code generation.



CONTENTS

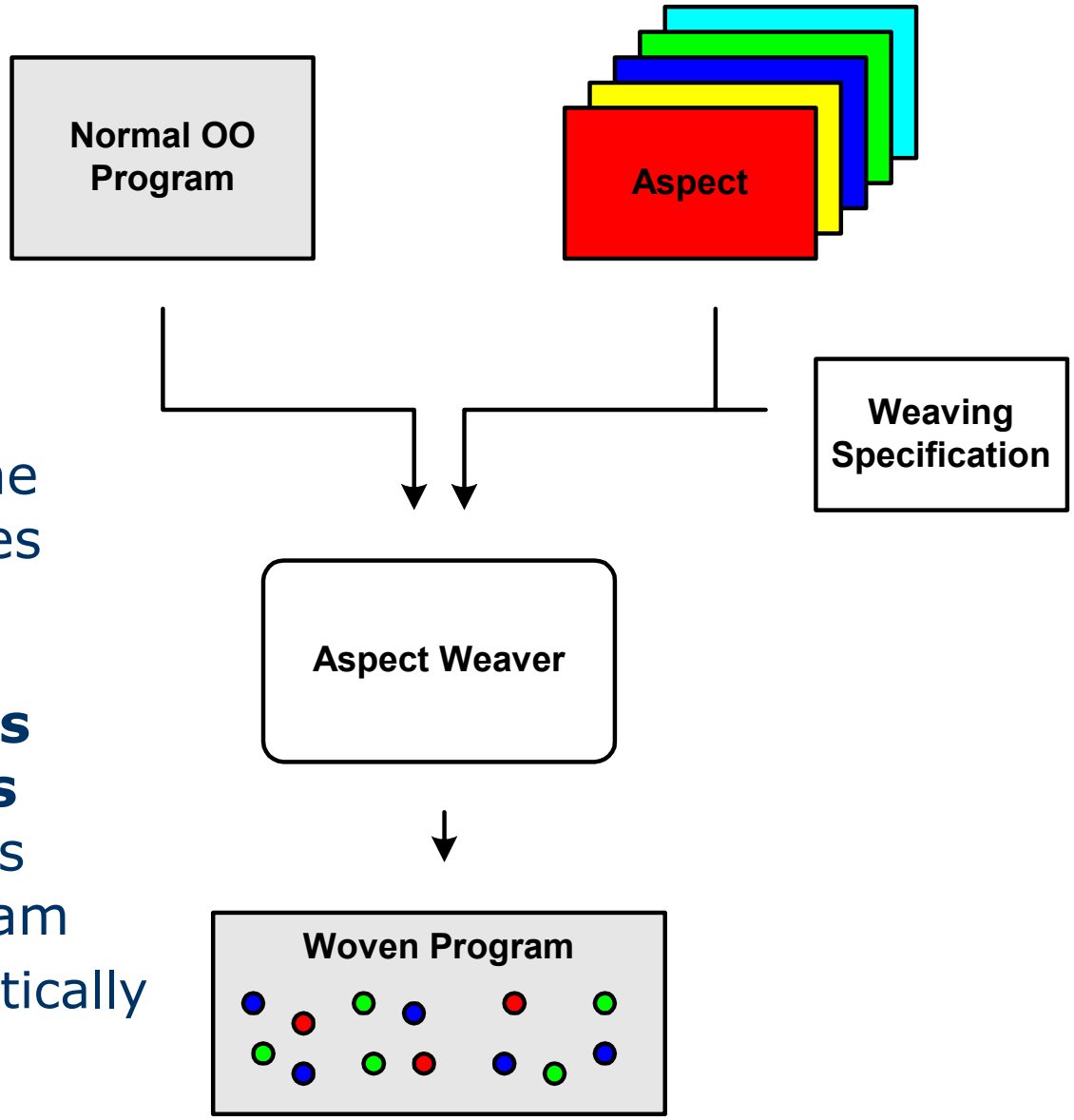
- What is MDSD
- **What is AOP**
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

What is AOSD?

- AOSD is about **localizing cross-cutting concerns** into well-defined modules called aspects.
- Various approaches to AOSD are possible, including **language extension** (AspectJ) and **framework/infrastructure-based** approaches (such as Spring AOP, JBOSS AOP or AspectWerkz).
- A core characteristic of each AOSD tool is its **join point model**, i.e. the means by which the base code and the aspect code can be joined.
  - **Static** and **Dynamic** join points can be supported
  - The **granularity** of the join point model varies.
  - **Introductions/Inter-Type declarations** are often, but not always possible

How does AOSD work?

- Developer develops **program code**
- Developer develops (or reuses) **aspect code**
- Developers specifies the **weaving rules** (defines pointcuts)
- Aspect Weaver **weaves program and aspects together** and produces the „aspectized" program
  - This may happen statically or dynamically



What AOSD is, too

- The above explanation of AOSD is what the **mainstream considers AOSD** to be.
- There are, however, two additional "aspects”:
  - **introductions**
  - and **collaborations**
- I will **not focus** on these in the main presentation – I will provide **some information at the end.**

CONTENTS

- What is MDSD
- What is AOP
- **Commonalitities and Differences**
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modeling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

Commonalities AOSD/MDSD

- **Separating Concerns.** Both approaches can be used to separate concerns in a software system.
  - AOSD typically modularizes CCC by separating them into aspects and later weaving them into the "normal" code (source or binary).
  - MDSD works by specifying system functionality in a more abstract, and domain specific DSL and the transformations are used to add those concerns that can be derived from the model's content.
- **Technical Aspects.** Both approaches are often used to factor out (and then later, reintegrate) repetitive, often technical aspects.

Commonalities AOSD/MDSD II

- **Mechanics.** Technically, both approaches work with queries and transformations
  - In AOSD you use pointcuts to select a number of relevant points (join points) in the execution of a program (or in its code structure) and "contribute" additional functionality called advice at these points.
  - In MDSD, a model transformation selects a subset (or pattern) of the model, and transforms this subset into some other model.
- **Metamodels.** Metamodels play an important role in both approaches.
  - In MDSD, the metamodel is clearly evident, as it forms the foundation of the model that is being transformed.
  - In AOSD the join point model of the particular AOP system is also a metamodel. A program execution is an instance of this metamodel.

Commonalities AOSD/MDSD III

- **Selective Use.** An important concept in both approaches is the fact that the handling of CCC can be turned on or off for a specific system.
  - In AOP, you can decide at weaving time whether you want to have a certain aspect included in the system.
  - In MDSD, you can select the transformation you want to use for a specific system – the chosen transformation may or may not address a certain concern.

Differences AOSD/MDSD

- **Dynamic vs. Static.**
  - MDSD works by transforming static models . That means, MDSD transformations work before the system is run at generation time.
  - AOSD, on the other hand, contributes behaviour to points in the execution of a system. In many systems it is therefore possible, to consider dynamic aspects in the definitions of pointcuts.
- **Invasiveness.**
  - MDSD needs to be used during the development of the software system, since the (finished) system is obtained by transforming models into code.
  - With AOP, however, it is (in most systems) possible to introduce behaviour after the base system has been developed completely.

Differences AOSD/MDSD II

- **Abstraction Level.**
  - A fundamental concept of MDSD is that it allows developers to express their intent with regard to the software system on a higher abstraction level, more closely aligned with the problem domain. A DSL serves exactly this purpose.
  - AOSD, on the other side, is basically bound to the abstraction level of the system for which it handles the CCC; in AOP, this is the base programming language.
- **Non-Programming Language Artefacts.**
  - In MDSD, it is easily possible to also generate non-programming language artefacts such as configuration files, build scripts or documentation.
  - AOP however works on the running system (remember it is dynamic in nature), and as such it cannot affect things that are not relevant at runtime
    - Exception: CME

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- **Forces**
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

The Topic of This Talk

- There are a number of commonalities between AOSD and MDSD. As a consequence, developers often don't know whether, or how they should relate AOSD and MDSD.
  - Should they use either AOSD or MDSD?
  - Is AOSD or MDSD a more general approach?
  - Is MDSD a special case of AOSD?
  - Or vice versa?
  - Can/should both approaches be used together, or would that just be "hype overkill"?
- The Problem is:

How can cross-cutting concerns be handled efficiently in an MDSD-based development environment?

Why not simply generate „AOP code“?

- Upon first look, one might think “**why not simply generate AspectJ** artefacts, why bother with all these alternatives”?

- Some reasons:
  - There **might not be a AO language** extension for the target programming language
  - We might want to modularize **non-programming-language** CCC artefacts
  - **Using an AO tool might not be possible** for technical, political or developer-skill reasons
  - Using an additional tool (an aspect weaver) in an MDSD environment adds **additional complexity** we might not want

- So it is well worth looking at alternatives…

How do we know? Forces

- **Applicability.** We would like to be able to use the pattern's solution to the problem above in as many situations, environments and "technology environments" as possible. The broader the applicability the better.
- **Granularity.** Different approaches provide different levels of join point granularity, over which pointcuts can be specified.
  - For example, an approach might only allow to advice calls to component operations,
  - whereas other approaches might allow interception of any method call in the system, thrown exceptions, field access, etc.

How do we know? Forces II

- **Performance/Footprint.** Each proposed solution has a more or less dramatic impact on system performance or footprint. In some environments, such as embedded systems, this can become a problem that deserves developers' attention.
- **Complexity.** While a certain approach solves a specific problem, it creates additional complexity – aka problems – in another area.
  - For example, the requirement to use additional languages or tools can be such an issue.
- **Flexibility.** Different approaches to CCC handling have different consequences with regards to (runtime) flexibility.
  - Some approaches allow to turn on/off the handling of a specific aspect at runtime or allow to change the behaviour at a certain pointcut, while others don't.

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - **Templates-Based AOP**
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

TEMPLATE INHERENT AOP // Context

- You are using a template-based code generator. The templates contain code that iterates over the model as well as textual output that should be created for a certain part of the model.
- The CCC you need to handle can be well localized in the templates.
- **Example.** This, creates a method signature and skeleton implementation for each Operation in a model.

```
«DEFINE OperationDef FOR Operation»
  public final «ReturnType» «Name» (
           «FOREACH Parameter AS p EXPAND USING SEPARATOR ", "»
             «p.Type.QualifiedJavaTypeName» «p.Name»
           «ENDFOREACH» ) {
   return «Name»Internal(
     «FOREACH Parameter AS p EXPAND USING SEPARATOR ", "»
              «p.Name»
     «ENDFOREACH» );
  }
«ENDDEFINE»
```

TEMPLATE INHERENT AOP // Solution

- Use normal template-level if statements to address the CCC. Depending on the if expression, a particular piece of code is either added to the generated code or not.
- **Example.** The following example code uses an if statement to add security checking in case security checks are enabled for the particular operation.

```
«DEFINE OperationDef FOR Operation»
  public final «ReturnType» «Name» ( … as before … ) {
    «IF checksRequired»
      if ( !Security.check("«Class.Name»", "«Name»") )
        throw new SecurityEx();
    «ENDIF »
    return «Name»Internal( … as before … );
  }
«ENDDEFINE»
```

TEMPLATE INHERENT AOP // Rationale

- A template is a meta program, a program that creates programs.
- As such, usually a number of base-level artefacts (here: operations) are created from a single template. If you need to handle concerns that cross-cut these locations, then a template modularizes this cross cutting concern.
- A simple if on template level is therefore enough to handle the CCC.

TEMPLATE INHERENT AOP // Consequences

- **Applicability.** Requires no special features in the target language. Not limited to programming language artefacts.
- **Granularity.** Can only be used if the pointcut is actually in a section of the code that is generated, pointcuts are limited to what is represented in the model, or to what can be derived by generation rules from the model.
- **Performance/Footprint.** There is no specific performance hit or footprint issue.
- **Complexity.** There is no complexity problem unless the same CCC cross-cuts the templates and not the generated code. This requires checking the if expression in multiple places. Good: You do not need additional (aspect) tools.
- **Flexibility.** The approach is completely static. Nothing can be changed (i.e. woven in/out) at runtime.

TEMPLATE INHERENT AOP // Known Uses, Summary

- From a tools perspective, every template-driven code generator can be used to implement this approach.
- As well, all MDSD projects I know of have used some form or another of this pattern to address CCC.
- This pattern is so ubiquitous, that mentioning specific known uses is pointless.

- **Summary.** While this approach seems rather trivial, it can be used to handle a surprisingly large amount of CCC that arise in practical work. And the fact that you don't need any AOP tool, is an additional benefit.

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - **AO Templates**
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

AO TEMPLATES // Context

- If you're building related families of code generators, using TEMPLATE-INHERENT AOP becomes too unwieldy because all kinds of concerns are handled inside the templates.
- **Example.**

```
«DEFINE OperationDef FOR Operation»
  public final «ReturnType» «Name» ( … as before … ) {
    «IF checksRequired»
      // security code
    «ENDIF »
    «IF loggingRequired»
      // logging code
    «ENDIF »
    «IF billingRequired»
      // billing code
    «ENDIF »
    return «Name»Internal( … as before … );
  }
«ENDDEFINE»
```

AO TEMPLATES // Solution

- Use an AO approach on template level. Rather that using template-level if statements, use an "aspect template" that advices the standard code generation templates with CCC-specific code.
- **Example.** The following piece of code defines two explicit join points: *MethodBegin* and *MethodEnd*.

```
«DEFINE OperationDef FOR Operation»
  public final «ReturnType» «Name» ( … as before … ) {
    «EXPAND HookMethodBegin»
    «ReturnType» res = «Name»Internal( … as before … );
    «EXPAND HookMethodEnd»
    return res;
  }
«ENDDEFINE»
```

- After these hooks have been defined, another template can attach itself to this hook. The following piece of code shows the logging aspect as an example.

AO TEMPLATES // Solution II

- **Example cont'd.**

```
«DEFINE LoggingMethodBegin FOR Operation AT HookMethodBegin»
    «IF loggingRequired»
      // entering method such and such
    «ENDIF »
«ENDDEFINE»

«DEFINE LoggingMethodEnd FOR Operation AT HookMethodEnd»
    «IF loggingRequired»
      // leaving method such and such
    «ENDIF »
«ENDDEFINE»
```

- The implicit scheme of defining join points means that "aspect templates" can attach to before or after already defined templates.

```
«DEFINE OperationLogging BEFORE OperationDef»
  // logging stuff
«ENDDEFINE»
```

AO TEMPLATES // Rationale

- This pattern basically introduces **AOP at the template level**.
  - TEMPLATE-INHERENT AOP uses normal template programming to handle CCCs in the resulting generated code by using ifs on template level.
  - This pattern handles CCCs on template level and uses AO techniques to address those.
- **Challenge:** For example, the following piece of code accesses the *Name* property of an Operation metaclass:

```
«DEFINE OperationDef FOR Operation»
  public void «Name» …
«ENDDEFINE»
```

- In order to access aspect-specific properties, we will have to add these properties to already existing metaclasses → **AOP on the metamodel**.

AO TEMPLATES // Consequences

- **Applicability.** Requires no special features in the target language. But **requires support from the generator** – not very widely provided!
- **Granularity.** Template-structure must provide a suitable structure
- **Performance/Footprint.** There is no specific performance hit or footprint issue.
- **Complexity.** In general, complexity is reduced by increasing modularity – same argument as for AO in general. Additional accidental complexity depends very much on the generator tool.
- **Flexibility.** The approach is completely static. Nothing can be changed (i.e. woven in/out) at runtime.

AO TEMPLATES // Known Uses, Summary

- The **openArchitectureWare** code generator [OAW] provides a feature called attached templates that implements this pattern.
  - It uses interceptors that can be configured by the developer to contribute additional operations to the metaclasses.
- Also, the **XVCL** frame processor allows to "contribute" frames (which can be seen as a form of code generation templates) to previously defined hooks.
- **Summary.** AOP on template level is very powerful.
  - However, the generator and its templates effectively become an AO language. The tools I am aware of only support this approach as an add-on, limiting the scalability of the approach.
  - Specifically, the IDE support that we are used to (such as AspectJ's Eclipse integration) is not available.

CONTENTS

AO PLATFORMS // Context

- You are generating code that is intended to run on a **technical platform**, usually some kind of communication or component/container **middleware**.
  - Such middleware typically already supports **factoring out some of the technical CCC** that occur in the domain for which the middleware has been developed.
  - The middleware platform usually also provides some kind of **configuration facility** to control how the middleware applies its CCC capabilities to the respective piece of application code.
- **Example.** In EJB systems, a component encapsulates functional (or domain) concerns.
  - Technical concerns such as transactions, security, load balancing, or pooling are taken care of by the **container**.
  - **Deployment descriptors** accompany the components and control how the application server handles them.

AO PLATFORMS // Solution

- Use the CCC-handling capabilities of the middleware as far as possible.
  - Use the code generator to **generate the annotations** that control how the middleware handles the (manually written, or generated) application code.
  - The information needed to generate the configuration is extracted from the model.
- **Example.** Many MDSD tools in the context of EJB require developers to develop POJOs that contain the business logic.
- The generator then creates “EJB wrappers” that ensure the POJOs conform to the constraints defined by EJB.
- The generator also creates a deployment descriptor to control the EJB container.

AO PLATFORMS // Rationale

- This pattern basically suggests to **leave the handling of the CCC to the target architecture**.
- To make this possible, the model needs to contain all the information that goes into the configuration.
  - We need to make sure this information does not clutter the "business" model described with our DSL. AO MODELLING is a good way to keep the core model clean.
- This pattern not just recommends to use a platform's CCC-handling capabilities in case it happens to provide these. Rather, **the pattern suggests to actively build** platforms that provide hooks to handle the typical CCC in the respective domain.

AO PLATFORMS // Consequences

- **Applicability.** The approach requires that the platform provides means to handle the CCC you need to address.
  - In case you build the platform for your domain, this is not a problem.
  - In case you use an off-the-shelf platform such as EJB, this can become a problem. Use PATTERN-BASED AOP or POINTCUT GENERATION.
- **Granularity.** Limited to the granularity provided by the platform.
- **Performance/Footprint.** Platforms that support the handling of CCCs will almost always imply some overhead, since it will always use some dynamic, generic, or reflective mechanism
- **Complexity.** The approach described in this pattern nicely factors out CCC into the platform. If the platform handles the CCC you need, this approach is unbeatable in simplicity.
- **Flexibility.** In case the platform handles CCC, it is usually possible to turn on/off a specific CCC during runtime, or change the way how the CCC is handled.

AO PLATFORMS // Known Uses, Summary

- **EJB** provides a platform where (some) CCC can be handled by specifying how they should be handled in the deployment descriptors.
- The **CORBA component model** (CCM) provides a similar feature. Plain CORBA allows developers to add interceptors to remote objects (or groups of remote objects, see.
- **Summary.** Non-trivial systems developed using MDSD will almost always include a rich, domain-specific platform, From a reuse perspective, it is a good idea to move as much (generic, domain-wide) functionality into this platform because you can use it from within the generated code.
- CCC are primary candidates for functionality in such a platform.

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - **Pattern-Based AOP**
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

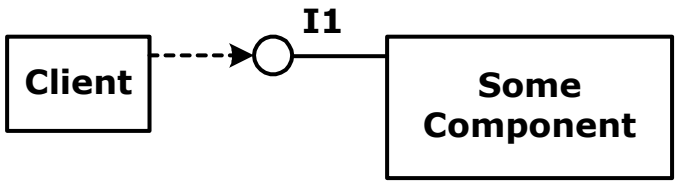
PATTERN-BASED AO // Context

- In some scenarios the platform you are required to use **does not provide services** that handle CCC, or it does not handle the CCC you need to address.
- You still need to have the flexibility to **change at runtime** the CCCs handled by the system.
- The pointcuts are accessible to the generation process.
- **Example.** Consider again an EJB based system. Consider also, that you need to implement so-called dynamic (or data driven) security. This means, you cannot use EJBs default (static) security model. However, you also don't want to bother application (component) developers with handling the security concerns.

PATTERN-BASED AO // Solution

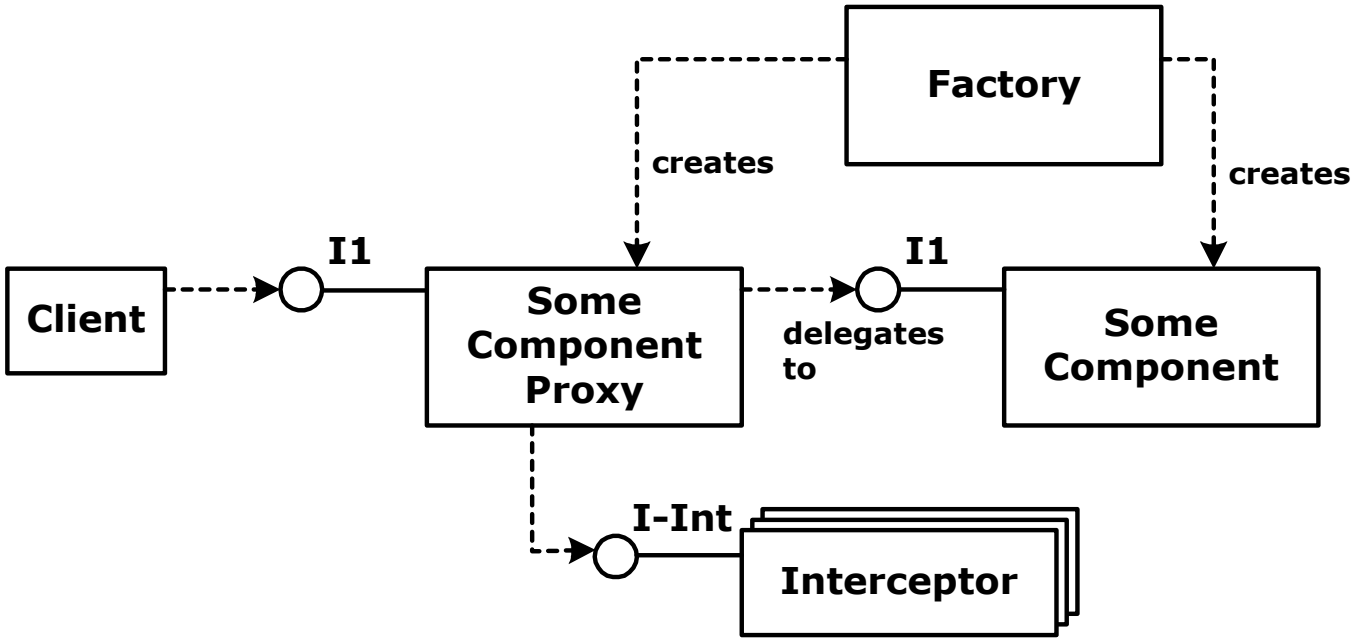
- Use a selection of the well-known patterns to generate an infrastructure that allows for custom CCC-handlers to be plugged in.
  - Typically, this consists of generating **proxies** [GoF] for application components
  - that can hook-in **interceptors** [POSA2].
  - Use a **factory** to instantiate the proxies if necessary.

- Consider you face the following situation:



PATTERN-BASED AO // Solution II

- You can replace this setup by the following:



- From a client's perspective, nothing has changed, the client still uses the interface I1. However, the client **actually talks to a proxy that handles CCC**, and then forwards to the real object.

PATTERN-BASED AO // Solution III

- Make sure that the join points are method calls; then the following **interceptor interface** can be used:

```
public interface Interceptor {
  public void beforeInvoke( Object target,
                            String methodName,
                            Object[] params );
  public void afterInvoke( Object target,
                           String methodName,
                           Object[] params,
                           Object retValue );
}
```

- The factory **determines which interceptors will be used** for a given object based on some kind of configuration (file).

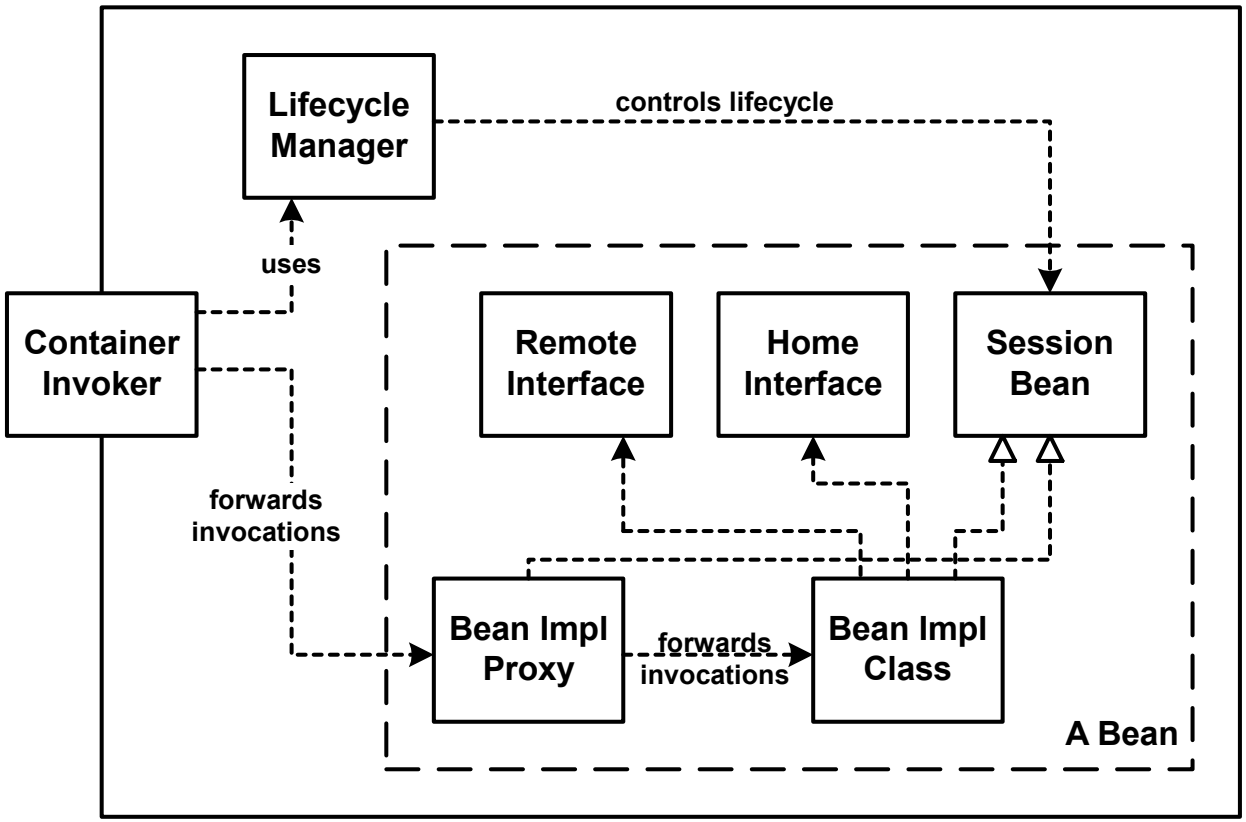
PATTERN-BASED AO // Solution III

- The following is the basic structure of the proxy:

```
public class SomeComponentProxy implements I1 {
  private SomeComponent delegate;
  private Interceptor interceptor; // can also be a list
                                   // of interceptors
  public String someOperation( String p1, int p2 ) {
    Object target = delegate;
    String opName = "someOperation";
    Object[] params = {p1, p2};
    Interceptor.beforeInvoke( target, opName, params );
    String res = delegate.someOperation( p1, p2 );
    Interceptor.afterInvoke( target, opName, params, ret );
    return res;
  }
  // more operations of I1
}
```

PATTERN-BASED AO // Solution IV

- **Example.** In the EJB scenario introduced above, the generated proxy would be the bean implementation class from the perspective of the application server, the real bean implementation would be an "implementation detail" of this class.



PATTERN-BASED AO // Rationale

- The approach here works whenever
  - (a) you can **influence object creation** so that the proxy is created instead of the real object, and
  - (b) if **method call-level pointcuts** are acceptable.
- This approach is **not really specific to MDSD**, you can in principle use the same approach in the context of normal, non-generative software development. However, since you would have to manually implement all the proxies, this is impractical, and thus hardly ever done.
- The **coarse granularity** seems like a limitation, in practice, it typically isn't.
  - In well-designed component based systems it is even desirable to apply aspects to component boundaries to keep the system manageable and understandable.

PATTERN-BASED AO // Consequences

- **Applicability.** The only precondition is that you are able to "tweak in" the proxy, which means generally, that you have to be able to control object creation. No specific features are required of the generator.
- **Granularity.** The approach only works for join points on method call level.
- **Performance/Footprint.** Considerable impact on performance
  - An additional object (the proxy) for each domain object.
  - For each method call, the method data has to be reified and the interceptor(s) called.
  - As a consequence, the approach does not really make sense for fine grained CCC. Using it on component level (as in the EJB example) is perfectly ok, though.
- **Complexity.** The approach does add complexity, since you have the proxies, the factories and the interceptors to deal with.
- **Flexibility.** Depending on whether the interceptors are configured at runtime or not, it is possible to add, remove or change "aspects" at runtime

PATTERN-BASED AO // Known Uses, Summary

- The approach to dynamic security in EJB has been used in several projects, some of which I have been directly involved with.
- A component infrastructure for small (mobile) devices implemented in Java uses the same approach.
- Java's Dynamic Proxy API uses the same idea, but based on reflection as opposed to static code generation.
- **Summary.** I have used this approach in various projects on component level and it works nicely. Particularly the EJB example above is useful (since it is completely portable and does not depend on and application server-specific features). Building the necessary generator (if you work with an MDSD approach anyway) is almost trivial.

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - **Pointcut Generation**
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

POINTCUT GENERATION // Context

- In some scenarios all the approaches described above don't work:
  - performance not sufficient
  - the platform does not support your needs,
  - or the granularity offered by the solution is too coarse.
- **Is there still hope?**
- **Example**. Component infrastructure for embedded systems
  - container is generated specifically for the scenario at hand
  - you want to add tracing, primarily for timing purposes.
  - Resource consumption and (near) real time behaviour is an important consideration, you cannot use generic solutions.
  - Cluttering the templates with all kinds of if statements is also not acceptable, because you need to trace different things at different times.

POINTCUT GENERATION // Solution

- Integrate an **AOP language** into the MDSD software development infrastructure.
  - Define a number of **pre-built advice**, part of the platform
  - **Generate the pointcut** based on specifications in the model.
  - **Use** the AOP language's **standard weaver** to integrate the aspects with the generated code.
- **Example.** XML below is part of the model for a node and container in the distributed, embedded system
  - Tracing option is set to *app,* i.e. we want to trace application level operations. (DSL-specific pointcut def.)

```
<node name="outside">
  <container name="sensorsOutside" tracing="app">
    …
  </container>
</node>
```

POINTCUT GENERATION // Solution II

- **Example cont'd.** As part of the platform, you define the following abstract aspect (using the AspectJ language). It does not define a pointcut, it is thus "pure advice".

```
package aspects;
public abstract aspect TracingAspect {
  abstract pointcut relevantOperationExecution();
  before(): relevantOperationExecution() {
    // use some more sophisticated logging,
    // in practice
    System.out.println( System.currentTimeMillis()+"::"+
                        thisJointPoint.toString() );
  }
}
```

- For every container that has tracing set to app, code like the following (a pointcut!) is generated:

```
package aspects;
public aspect SensorsOutsideTrace extends TracingAspect {
  pointcut relevantOperationExecution() :
    execution( * manual.comp.temperatureSensor..*.*(..) ) ||
    execution( * manual.comp.humiditySensor..*.*(..) );
}
```

POINTCUT GENERATION // Solution III

- **Example cont'd.** This generated aspect can now be woven with the rest of the (generated, or manually written) code, and thus add tracing to the required parts.
- See below for the code generation template

```
«DEFINE TracingAspect FOR System»
  ...
  «FOREACH Container AS c EXPAND»
    «IF c.Tracing == "app"»
      «FILE "aspects/"c.Name"Trace"»
      package aspects;
      public aspect «c.Name»Trace extends TracingAspect {
        pointcut relevantOperationExecution() :
          «FOREACH c.UsedComponent AS comp
                      EXPAND USING SEPARATOR "||"»
             execution(* manual.comp.«comp.Name»..*.*(..) )
          «ENDFOREACH»;
      }
      «ENDFILE»
    «ENDIF»
  «ENDFOREACH»
  ...
«ENDDEFINE»
```

POINTCUT GENERATION // Solution IV

- If your base language as well as the AOP language extension support **metadata annotations** (for example, AspectJ 5 or a combination of Java 5 and AspectWerkz) you can use the following approach:
  - The pre-built aspect includes an pointcut definition that **tests the presence of a certain metadata attribute**.
  - If it is present, **the artefact is selected by the pointcut**, and the advice is added.
  - The **code generator** simply has to **add the metadata attribute to the artefact**, if it wants the artefact to be affected by the advice.
- Note that in languages that don't support annotations, you can **alternatively use marker interfaces** – although this only works for advising classes, and not other artefacts such as fields or operations.

POINTCUT GENERATION // Rationale

- An interesting question is **how to integrate AOP languages** efficiently.
  - Providing advice as part of the platform and then generating the pointcuts is a very useful approach indeed.
  - This requires that the domain developer decides which advice might be necessary.
  - However, this is required anyway, since the DSL has to have a feature to control where to apply the aspect, and where not.
- One could also **specify the tracing concern in a separate model** (see AO MODELLING). Example:

```
<trace-config>
  <trace container="sensorsOutside" level="app"/>
  <trace container="sensorsInside" level="all"/>
</trace-config>
```

POINTCUT GENERATION // Rationale II

- It is interesting to see that this pattern suggests using an **AOP language** such as AspectJ **as an implementation technology** in MDSD projects.
  - The aspectual nature of the tracing concern does not show up in the DSL.
  - Rather, AOP is used to keep the implementation of the tracing feature small and fast.
- I think that this is the primary use case for languages like AspectJ in the context of MDSD.

POINTCUT GENERATION // Consequences

- **Applicability.** The approach can be applied only if, for the respective target language, an AO extension is available.

  There are no special requirements for the generator.

- **Granularity.** The achievable granularity depends on the join point model of the aspect language used.
- **Performance/Footprint.** Depends very much on the implementation of the aspect language, specifically, when the weaving occurs.
- **Complexity.** Complexity can raise significantly using this approach, since it opens up a “whole new can of worms”.
- **Flexibility.** Again, this depends on the aspect language used for the implementation.

POINTCUT GENERATION // Known Uses, Summary

- The small components prototype uses this approach to handle CCCs that cannot be handled using PATTERN-BASED AOP.
- In the context of mobile phone software, the pattern has been used to generate static aspects (aspects that produce compile time errors) to check developer conformance to programming guidelines.
- **Summary.** This approach is certainly the most powerful.
  - However, it requires the use of an aspect language in addition to all the generator and modeling tools that are necessary for MDSD anyway.
  - This can be a huge problem in practice.
  - Also, in contrast to the MDSD approach, most AO language extensions require runtime support libraries. In some production environments (such as in large companies) this can be a showstopper

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - **AO Modelling**
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

AO MODELLING // Context

- Up to now, we were mainly concerned with **handling CCC in the resulting application**, which would be built using an MDSD approach.
  - App is described using models, and model transformations and code generation is used to create the final application.
- In many scenarios, however, it is necessary to **separate concerns in the application models**, too!
- **Example.** Consider you are building a web application. Such a web application typically consists of
  - (a) a business object model,
  - (b) the persistence mapping of this model,
  - (c) the web pages, forms and the workflow, and
  - (d) the layout of these forms and pages.
- You have to **specify all this in the model** in order to be able to generate a complete application.

AO MODELLING // Solution

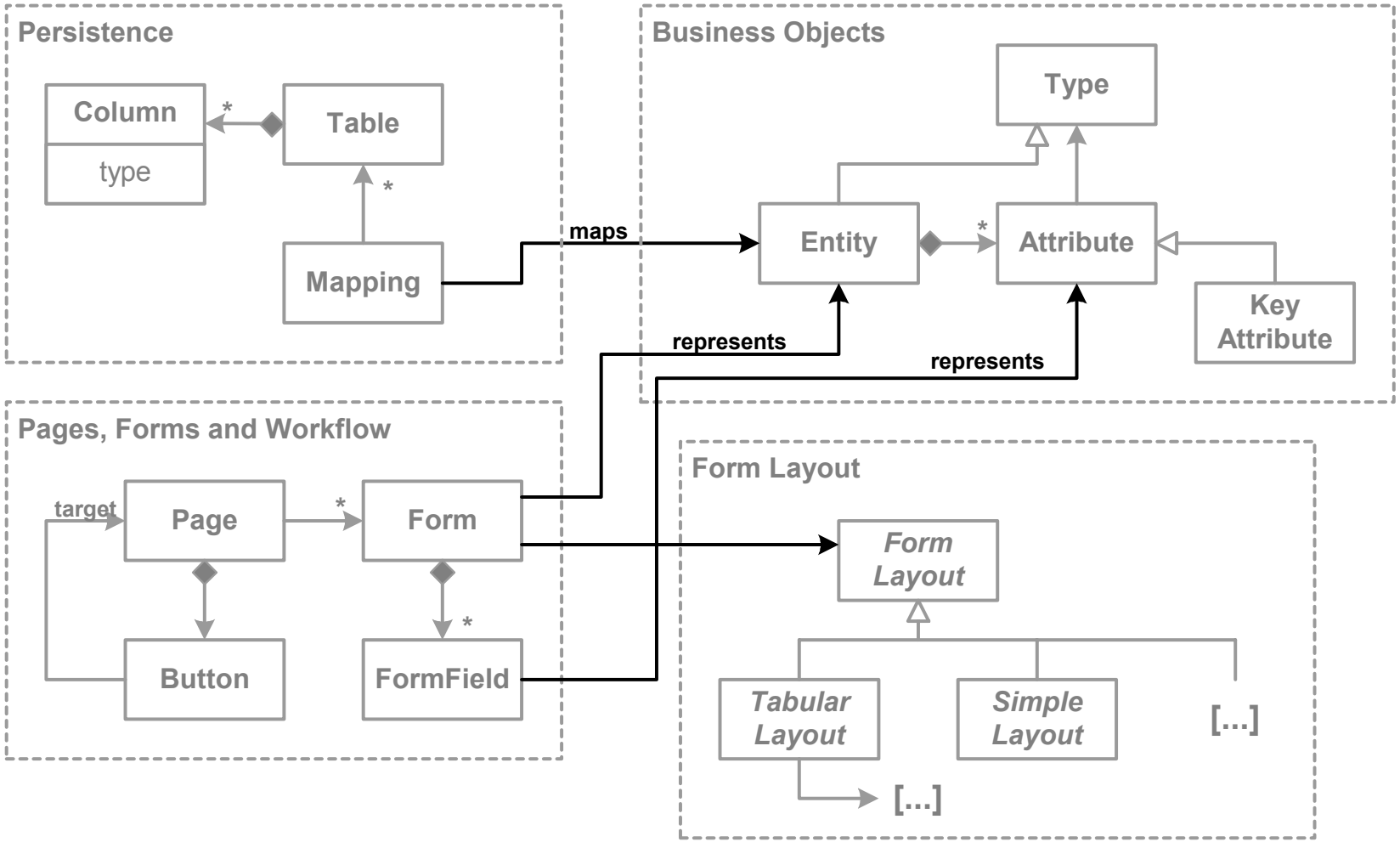
- Create several models, **one for each aspect**.
  - Each model uses **a DSL** (i.e. concrete syntax and metamodel) suitable for the expression of the particular aspect.
  - The **code generator** reads all these models, **weaves them**, and then generates the complete application from it.
  - Join points are defined on the metamodels, for example, by **using a specific metaclass in more than one aspect's metamodel**, thereby building up links between the models.

AO MODELLING // Solution II

- **Example.** In the example above, you could use
  - (a) a **UML class diagram** (of course, with suitable stereotypes) to describe the business object model,
  - (b) **an XML** document to describe tables and the mapping,
  - (c) **another class diagram** (with other stereotypes) to describe pages, forms and the workflow, and finally,
  - (d) a **Visio diagram** to describe form layout.
- Each of these **four models** need to be connected suitably to describe a complete and consistent system.
- For this to work, the **metamodels must be related**, as shown in the next illustration. Note how associations cross the various aspect metamodels.

AO MODELLING // Solution III

- **Example cont'd.** Metamodels and their connections.



AO MODELLING // Rationale

- This pattern effectively suggests to have **a separate model for each aspect**. The challenge of this approach lies in the fact that the generator tool must be able to:
  - read the various models
  - check for consistency
  - weave the models
- Note that you **cannot have separate generators** for the different aspects, since the metamodels (and thus, also the models) are related.
- Note that implementing this weaving process is **much easier inside a generator** compared to a tool like AspectJ.
  - The reason is, that your domain metamodels are usually vastly simpler than the metamodel (i.e. abstract syntax) of a language like Java,
  - and that you define the join point model yourself.

AO MODELLING // Consequences

- **Applicability.** The approach can be applied only if the generator can handle various models with different concrete syntaxes and is able to perform the weaving.
- **Granularity.** The achievable granularity is completely under the control of the developer, since the join point model is part of the (custom) metamodel definition.
- **Performance/Footprint.** This pattern has no consequences for runtime performance and footprint, since it is applied at generation time.
- **Complexity.** If your generator really represents all models as objects in the implementation language once they are parsed, the implementation of this pattern becomes almost trivial.
- **Flexibility.** This issue does not apply here, since the pattern has no runtime consequences.

AO MODELLING // Known Uses, Summary

- **All MDSD projects that I am or was involved in** have used this approach,
  - this includes a C-based component model for embedded real time systems,
  - web applications
  - and components for mobile devices.
- The **documentation of the openArchitectureWare generator** shows an extensive practical example of using more than one model as generator input.
- **Summary.** In non-trivial scenarios, AO MODELLING is absolutely necessary to keep (large) models manageable.
  - Make sure you use a tool where this approach can be implemented painlessly, before you use the generator tool on larger projects.

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- **Pattern Relationships**
- Introductions and Collaborations
- Overview and Summary

PATTERN-BASED AOP and AO PLATFORMS

- PATTERN-BASED AOP basically combines a couple of design patterns to **implement an interception framework**. The necessary proxies are created using code generation.
- You can "morph" this pattern to become an AO PLATFORM in the following way:
  - use **runtime code generation to add the necessary proxies** (or more general, hooks) to the system, for example during class load time.
  - use a configuration file to **define which interceptors should be used** for a certain class.
- You can then use this infrastructure as the AO PLATFORM for your application code.

AO PLATFORM and POINTCUT GENERATION

- The boundaries between AO PLATFORMS and POINTCUT GENERATION seems to blur. There are clearly the two extremes:
  - **AspectJ is an AO language** extension for Java. Using it is definitely an instance of the POINTCUT GENERATION pattern.
  - **EJB 2.x are a – quite limited – AO PLATFORM**. Deployment descriptors allow you to handle certain predefined CCC.
- JBoss AOP, for example, is **not as readily categorized** into one of these two categories. You can define arbitrary advice (basically, by implementing interceptors) and then define a pointcut definition in a separate XML file.
  - **On the one hand** it is POINTCUT GENERATION: you generate a pointcut (the XML file) that determines where to weave in pre-built advice (the interceptors).
  - **On the other hand** it is an AO PLATFORM, since it also comes with a set of predefined advice that are typically used in the relevant domain (enterprise systems).

The special case of AO MODELLING

- AO MODELLING **plays a somewhat special role** in that it can be used together with any of the other patterns, since it takes care of CCC on the "input side" of the MDSD process.
- You **cannot substitute this pattern** by using an AOP language extension in the generated code.

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- **Introductions and Collaborations**
- Overview and Summary

Introductions

- Introductions are used **to "inject" artefacts into an existing system statically** (as opposed to dynamic advice in join points).
- For example, additional fields or operations can be introduced into existing classes.
  - An example could be "for all classes that implement interface X, add the following methods:").
- Some AOSD languages also allow to **change static class features**, such as changing the superclass, or adding an additional implemented interface; this is called **open classes.**

Introductions II

Pattern	Introductions	Open Classes
TEMPLATE-INHERENT AOP	Using a template-IF, it is easy to conditionally inject code into the generated artefacts.	
AO TEMPLATES	It is possible to add template code to existing templates. The explicitly defined hooks shown above can be considered to be an introduction.	N/A
AO PLATFORMS	Depends on the platform, not widely supported. Some allow it by using tools such as byte-code modification (Spring is an example).	
PATTERN-BASED AOP	Not possible.	
POINTCUT GENERATION	Depends on the AOSD tool used. If you use AspectJ, for example, both features are possible.	
AO MODELLING	Using on the fly model modifications, it is common practice to add additional features to model elements.	N/A

Collaborations

- A **collaboration between artefacts** can be considered an aspect.
- Using this approach, **a collaboration becomes a type**, in the same way as classes or aspects (as we know them traditionally) are types.
- Using this approach, you can **capture large portions of collaboration code in well-separated aspects** and then simply bind concrete artefacts to instances of these collaborations. "Traditional" AOSD mechanics are used to fill in the required "magic".
- This "kind" of AOSD is very important for **handling functional aspects** as opposed to technical aspects but is still subject for research and not widely used in practice.

Collaborations II

- General Idea of collaboration aspects:
  - **Define a collaboration** as a type.
    for example, the Observer pattern
  - The collaboration type **describes the various roles** that are required for the collaboration
    in the example, Subject and Observer
  - In a concrete system, pointcuts are used to **instantiate the collaboration by binding** concrete artefacts to the instantiated collaboration
    for example, in a drawing program, the class Figure plays the role of the subject, while the Canvas plays the role of the Observer.
  - The collaboration also defines, **which features artefacts must have** in order to be able to play a role
    for example, they must implement the ISubject interface, or have an operation registerObserver().

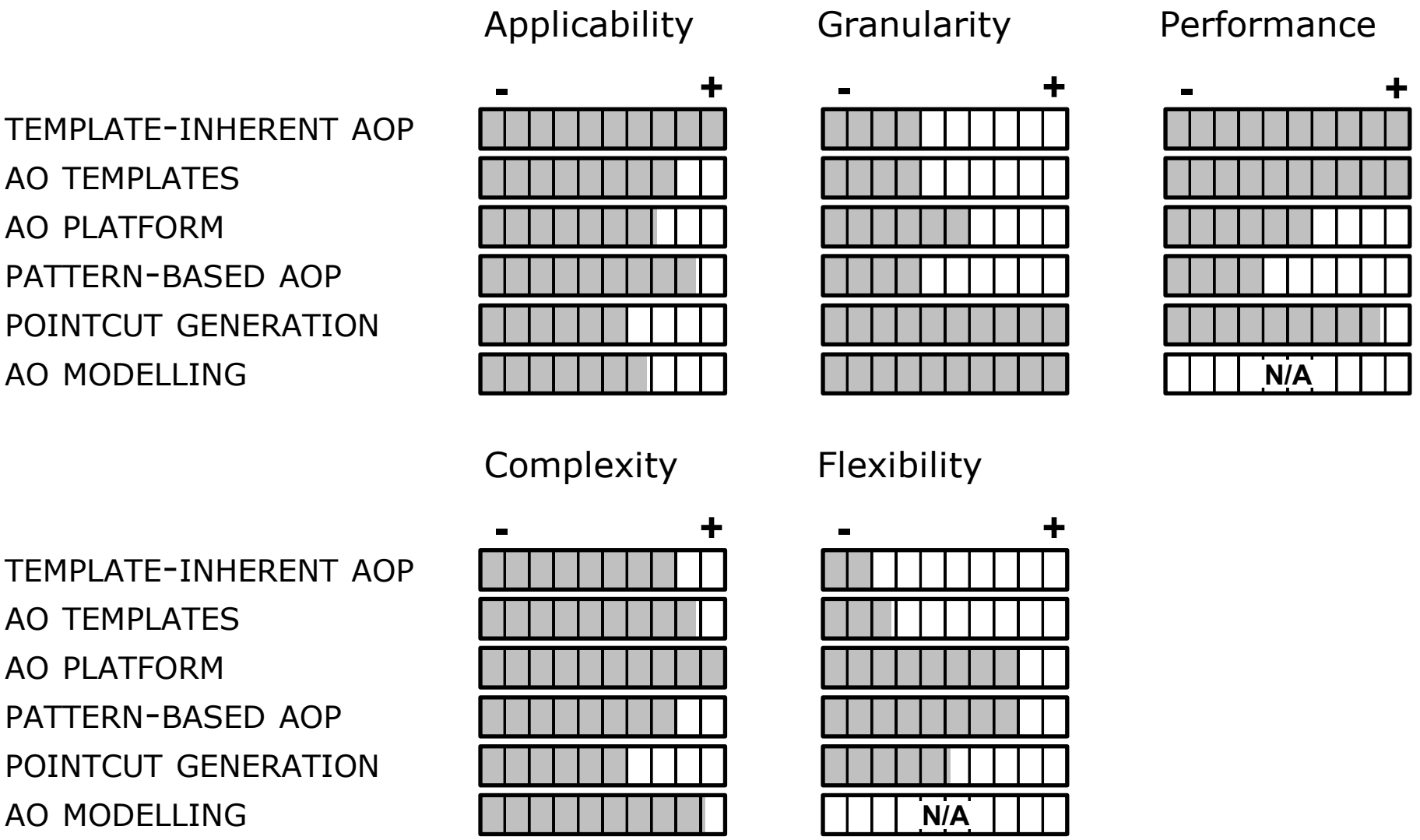
Collaborations III

- The **patterns do not address this aspect**, with two notable exceptions:
  - The AO Moelling pattern can support this approach nicely:
    - **Use markup in models** (such as tagged values in UML models) to mark certain artefacts as playing a certain role in a collaboration.
    - The generator can then make sure **the model artefact has all the required features**, or use model modifications to actually add them.
    - Later stages can make sure the **generated code can play** the collaboration role.
  - You can extend the POINTCUT GENERATION pattern so that you **generate collaboration bindings** in case the underlying AOSD language supports this.

CONTENTS

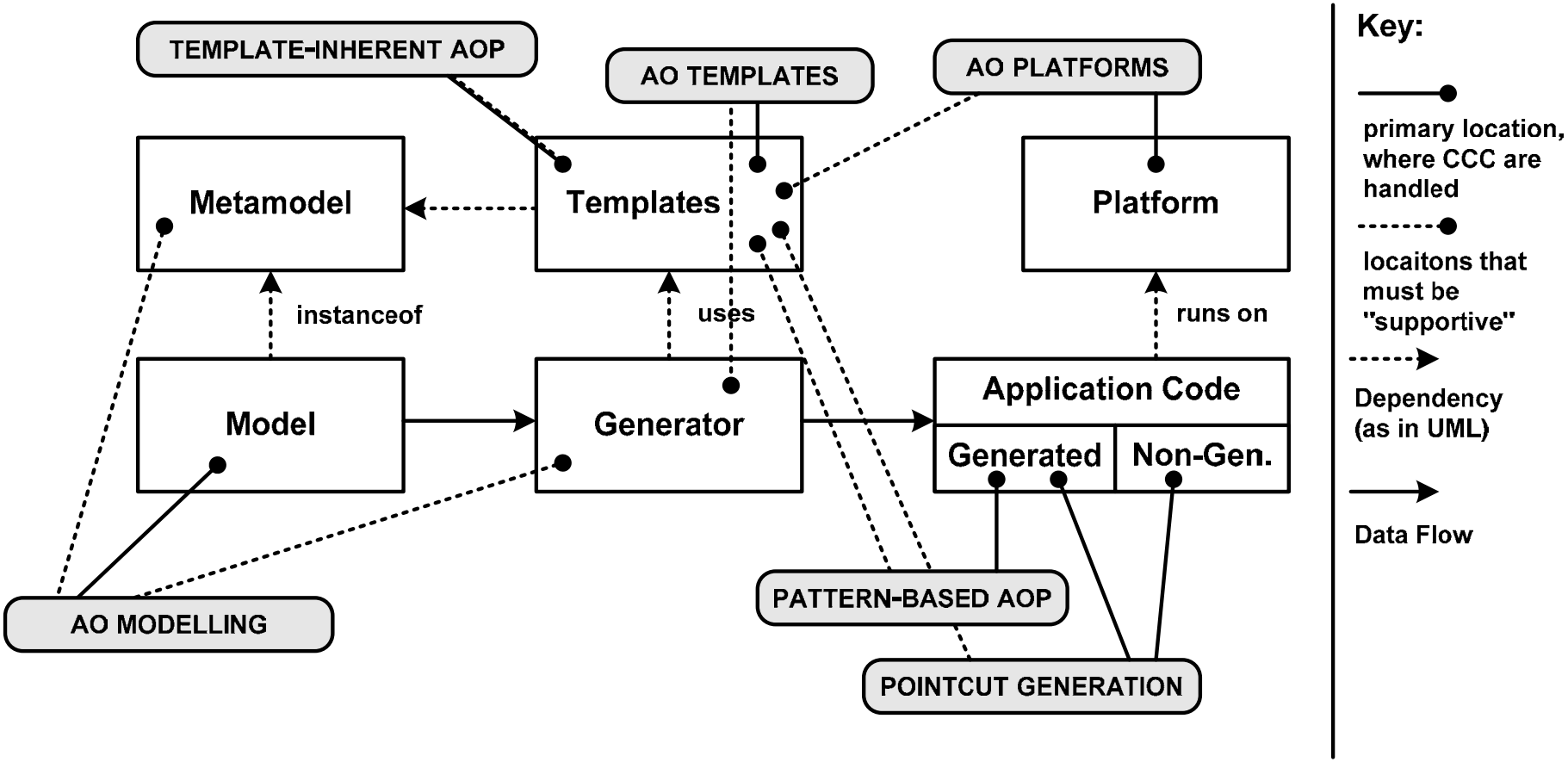
- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- **Overview and Summary**

Overview of the Consequences



Overview of the Approaches

- This illustration shows where is an MDSD workflow CCC can be handled.



Where is the loom?

Pattern	Weaving Location
TEMPLATE-INHERENT AOP	No weaving happens – the advice are inlined into the template code.
AO TEMPLATES	The weaving is handled by the template engine.
AO PLATFORMS	The platform takes care of weaving, typically at load time or runtime
PATTERN-BASED AOP	The generator creates the proxies during system generation. Adding the interceptors (i.e. defining pointcuts) can happen during system startup or at any time during runtime.
POINTCUT GENERATION	The pointcuts are generated statically. The weaving happens in a separate weaving phase, at load time or at runtime, depending on the used AO tool.
AO MODELLING	The weaving is done by the code generator (acting as a model weaver) *before* code generation.

CONTENTS

- What is MDSD
- What is AOP
- Commonalitities and Differences
- Forces
- Patterns
  - Templates-Based AOP
  - AO Templates
  - AO Platforms
  - Pattern-Based AOP
  - Pointcut Generation
  - AO Modelling
- Pattern Relationships
- Introductions and Collaborations
- Overview and Summary

THE END.

Some advertisement ☺

- Völter, Stahl
- **Modellgetriebene Softwareentwicklung**
  Technik, Engineering, Management
- dPunkt 2005
- www.mdsd-buch.de