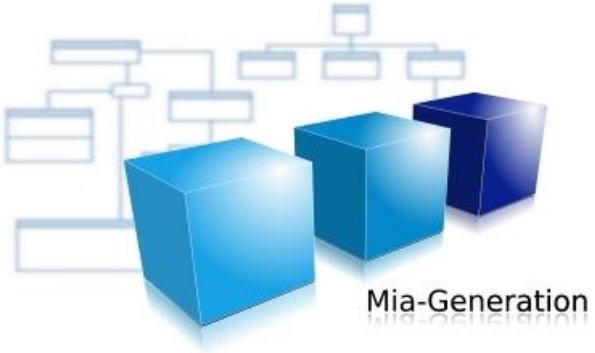
Mia-Software

IT Agility Solutions

Introduction à l'activité de Génération de code

Formation



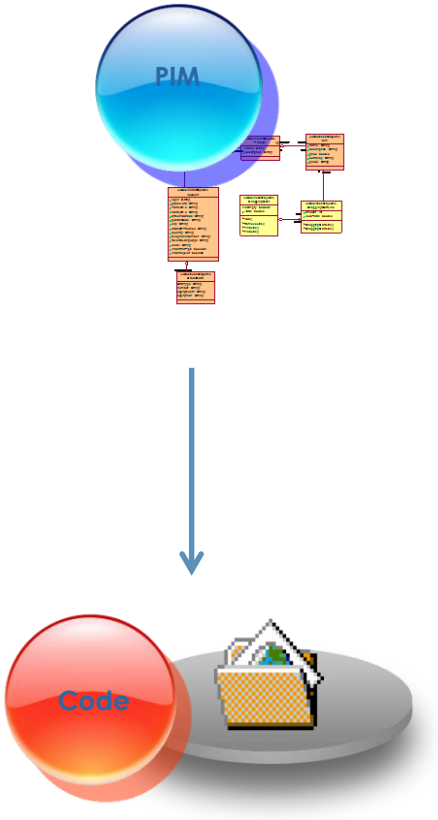
1.0

Sommaire

- Concepts & Acteurs
- Ecriture de générateur de code
  - Langages
  - Ateliers
- Intégration d'un générateur dans une usine
  - Besoins
  - Cycle itératif génération/développement

Génération de code

- Définition dans le cadre MDA (www.omg.org/mda/)
  - Automatiser partiellement le passage d'un niveau de modèle au code
    - PSM -> Code
    - ou PIM (+ PDM) -> code
- Apports
  - Décrire un logiciel indépendamment de son implémentation
    - ✓ **Pérennité**
  - Déduire le code par traduction automatique du modèle
    - ✓ **Productivité & Qualité**
  - Adapter les règles de traduction au gré des évolutions techniques
    - ✓ **Agilité**

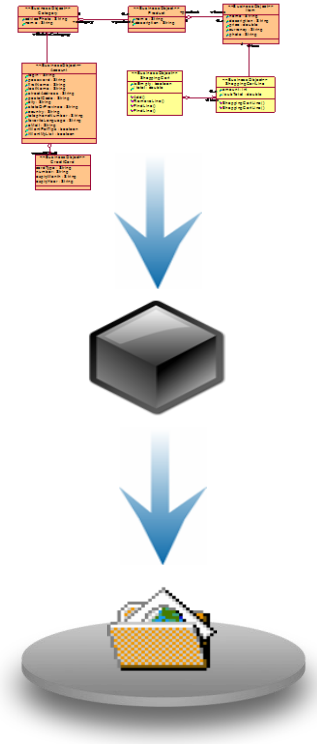


Concepts & Acteurs

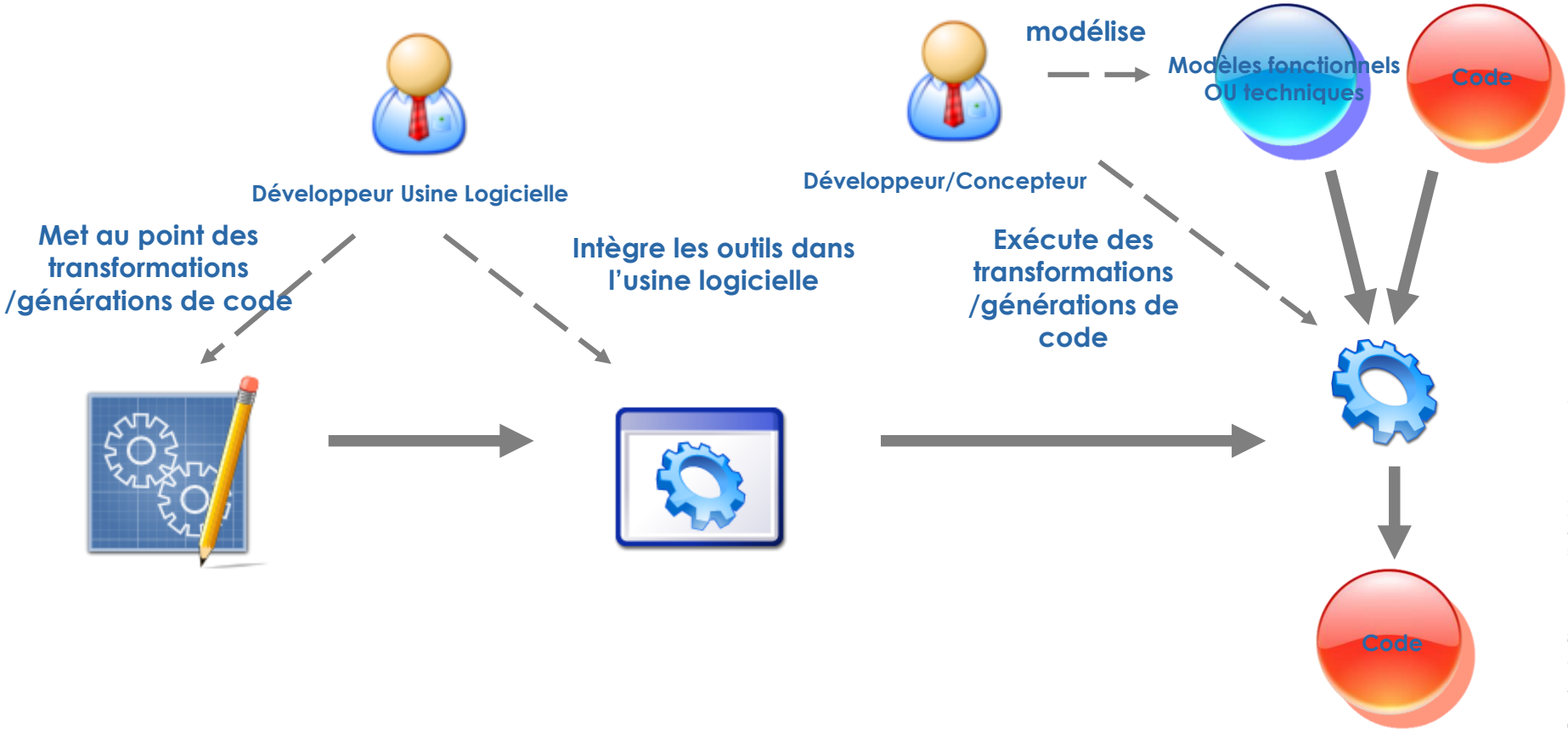
- Illustrations d'usines logicielles incluant génération de code (clients Mia-Studio)

Modelisation	Plateforme Technique Cible	Domaine	Remarques
UML2 (MagicDraw UML)	Persistence Hibernate/Oracle Backoffice en Java. Front Office en Eclipse RCP, Communication en webservices.	Aéronautique	Taux Generation 100% (presentation + backoffice)
UML2 (MagicDraw UML)	Persistence Hibernate JEE Front Office en Web JSF	Banque/Assurances	
MCD (Power AMC)	Couche Metier Cobol Front Office en Web HTML+JS	Banque/Assurances	
UML2 (RSA)	Java	Aéronautique	
UML2 (MagicDraw UML)	Couche Metier JEE (EJB) - Spring + .Net Communication en Web Services Front Office JEE-Struts + .Net (.aspx)	Banque/Assurances	
UML1 (Rose)	Java	Banque/Assurances	100 000 fichiers générés
UML2 (MagicDraw UML)	Java Front Office en Flex	Banque/Assurances	Separation fichiers générés (Interfaces) et code dévéloppeur (Implémentations)
UML2 (RSA)	JEE Couche communication en Web Services		Generateur = 10000 scripts, 6 générateurs, 10 lancements par mois, sur 300 postes, volume de code généré important (Java, XML)
UML2 (MagicDraw UML) + UML1 (Rose)	C++ Generation Documentaire openoffice/Word/html	Aéronautique	
UML2 (MagicDraw UML)	Persistence Hibernate JEE - Spring Front office en Flex	Services	Generateur = 7000 scripts, 46000 lignes de code Taux de génération 60% de code généré 10% de code manuel dans fichiers générés 30% de fichiers non générés

- Historique
  - La génération de code existe depuis l'origine de l'informatique
    - Un compilateur est un générateur de code : il produit du code exécutable à partir d'un langage de plus haut niveau que le langage machine
  - 1980/2000 : montée en abstraction et génération de code non paramétrable « Boîte noire »
    - Ex: PACBASE (génère du COBOL) PowerBuilder (génère du C++)
    - Ex: Together de Borland représentait le code en UML et réciproquement
  - > 2000 : outils de génération sur mesure « Boîte blanche »
    - MDA &MDD(model-driven architecture & development), DSL (domain specific language)
- Ecueils d'une approche de génération « boîte noire »
  - Qualité du code généré
    - Pas de respect des normes de codage de l'entreprise
  - Intégration des évolutions techniques
    - Dépendance complète vis-à-vis de l'éditeur pour intégrer des nouveautés technologiques
  - Souplesse méthodologique
    - Pas de prise en compte de la méthode d'analyse/conception de l'entreprise



Concepts & Acteurs



Sommaire

- Concepts & Acteurs
- Ecriture de générateur de code
  - Langages
  - Ateliers
- Intégration d'un générateur dans une usine
  - Besoins
  - Cycle itératif génération/développement

- Approche « Programmative »
  - Construction de fichiers par utilisation d'APIs d'accès aux modèles

```
if(attributeAbstract.equals("false") && !childElementSet.contains(attributeName)) {
  pw.println(" <xs:element name=\""+attributeName.toLowerCase()+"\ttype=\"xs:"+attributeType+"\"");
  pw.println(" <xs:annotation>");
  pw.println(" <xs:documentation>");
  BufferedReader br = new BufferedReader(new StringReader(attributeComment));
  String ss;
  while((ss = br.readLine()) != null) {
    pw.println(" "+ss);
  }
  br.close();
  pw.println(" </xs:documentation>");
  pw.println(" </xs:annotation>");
  pw.println(" </xs:element>");
}
```

- Approche « Template »
  - Construction de fichiers par fragments représentatifs (WYSIWYG)
  - Exemples de langages : Velocity, Jet, MOFM2T

```
</xsd:documentation>
</xsd:annotation> <xsd:sequence>
#foreach ($superType in $type.allGeneralizations)
#foreach ($attribute in $superType.attributes)
  <xsd:element
      name="$attribute.name"
      nillable="$attribute.nillable"
      type="$attribute.type.schemaType">
    <xsd:annotation>
      <xsd:documentation>
        $attribute.getDocumentation(" ", 64, false)
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
#end
#end
```

Sommaire

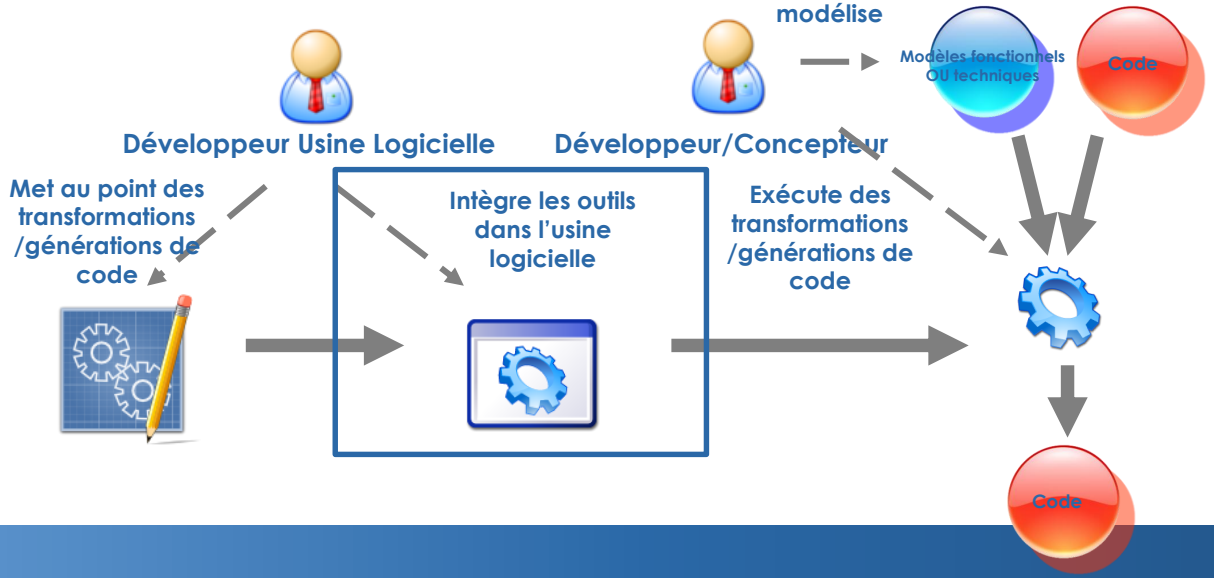
- Concepts & Acteurs
- Ecriture de générateur de code
  - Langages
  - Ateliers
- Intégration d'un générateur dans une usine
  - Besoins
  - Cycle itératif génération/développement

- Caractéristiques et critères de choix d'un atelier de génération de code
  - Langages : Combiner les approches API et Templates
  - Atelier : caractéristiques classiques d'un AGL
    - Editeurs dédiés aux langages (complétion, coloration syntaxique, …)
    - Mise au point (traces interactives, points d'arrêt, …)
    - Approche par composants réutilisables
    - Possibilités de travail collaboratif sur les générateurs
    - Possibilités de construction & non régression des générateurs en intégration continue
    - Extensibilité de l'atelier
    - …
- Exemples Ateliers
  - Mia-Studio
  - Acceleo (MOFM2T)
  - Jet
  - Xpand/Xtend

Sommaire

- Concepts & Acteurs
- Ecriture de générateur de code
  - Langages
  - Ateliers
- Intégration d'un générateur dans une usine
  - Besoins
  - Cycle itératif génération/développement

- Intégration dans l'usine logicielle : Besoins & Critères de choix d'atelier
  - Compatibilité avec les ateliers de modélisation
  - Déploiement & Utilisabilité au plus près de l'atelier du développeur final
  - Pilotage silencieux de génération (API, ligne de commande, …)
  - Support de cycle itératif génération/developpement

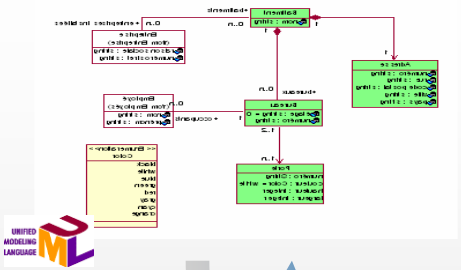


Sommaire

- Concepts & Acteurs
- Ecriture de générateur de code
  - Langages
  - Ateliers
- Intégration d'un générateur dans une usine
  - Besoins
  - Cycle itératif génération/développement

Cycle itératif classique de génération/développement de code d'une application

Modélisation



Itération de modélisation

Concepteur

Itération de génération

Génération de code

```
// Start of user code for import
// End of user code for import

public final class Address {

    private String city;
    private String country;
    private int number;
    private String street;
    private String zip_code;

    // ----- CONSTRUCTORS -----
    // Start of user code for Constructors.
    // End of user code for Constructors.

    public Address() {
        // Start of user code for default constructor
        // End of user code for default constructor
    }

    // ----- ACCESSORS -----
    // Start of user code for Accessors.
    // End of user code for Accessors.

    // ----- Accessors for the city attribute -----
    public String getCity() {
        return this.city;
    }

    public void setCity(String element) {
        this.city = element;
    }
```

Java

Itération de développement Java

Développeur

Déploiement

Travail du développeur

- La génération ne produit pas 100 % du code final en général
  - Nécessité d'intervention de développeurs pour les compléments

  ("taux de génération" = pourcentage du code généré/code final)
  - Nécessité de protection du code "généré" et du code "développeur"
- Approches classiques
  - répartition du code « généré » et « développeur » par fichiers
  - séparation du code « généré » et « développeur » dans un fichier par zones balisées ou marquage

```
public class Employee {

    // ----- ATTRIBUTES -----
    // Start of user code for Attributes.
    // End of user code for Attributes.

    private Address address;
    private Company company;
    private Division division;
    private String first_name;
    private List<Division> managed_divisions = new ArrayList<Division>();
    private String name;
    private Office office;
    private Map<String, Phone> phone = new HashMap<String, Phone>();
    private Rank rank;
```

Développeur

```
public class Employee {

    // ----- ATTRIBUTES -----
    // Start of user code for Attributes.
    private String myTechnicalInstanceVariable;
    // End of user code for Attributes.

    private Address address;
    private Company company;
    private Division division;
    private String first_name;
    private List<Division> managed_divisions = new ArrayList<Division>();
    private String name;
    private Office office;
    private Map<String, Phone> phone = new HashMap<String, Phone>();
    private Rank rank;
```

Retour d'expérience

- Vision initiale de la génération de code dans le cadre MDA
  - Le système de balises associé à une rigueur dans l'application du processus devait permettre d'apporter tous les gains espérés (productivité, qualité, …)
- La réalité : productivité & qualité ne sont pas toujours au rendez-vous
  - Non respect des zones balisées par manque de formation des développeurs
  - Délai élevé de remontée d'un besoin au niveau modèle avant nouvelle génération (problème organisationnel ou défaut d'intégration des outils)
  - Non respect délibéré des zones balisées (rejet par le développeur)
- Conclusion
  - Privilégier la séparation des fichiers générés (quitte à un volume généré moindre)
  - Consacrer du temps à la formation et à l'écoute du développeur qui finalise le code