Towards agile cross-platform application development with Smalltalk and Model Driven Engineering

[ab]Glenn Cavarlé [a]Alain Plantec [a]Vincent Ribaud [b]Christophe Touzé

[a]Lab-STICC, UMR CNRS 6285, Université de Bretagne Occidentale, UEB, 20 av. Le Gorgeu, Brest, France
[b]SARL Diazol, 156 rue Jean Jaurès, Brest, France
{glenn.cavarle,alain.plantec,vincent.ribaud}@univ-brest.fr, christophe.touze@diazol.com

Abstract

Nowadays, general public applications or specific information systems must be able to run on mobile platforms as well as on conventional platforms. Because developers have to deal with mobile platform specificities, this constraint significantly lessen the benefits of agile methods and as a consequence impacts on the application development cost. Many research works aim at reducing the development cost. Prototyping as well as automatic code generation have been investigated by the community. In this article, we present Dali, a framework that uses both Smalltalk and Model Driven Engineering. With Dali, an application model can be designed for multiple platforms and interpreted before code generation. An execution platform is modelled as a set of constraints over a context. These constraints can affect the presentation of the Graphical User-Interface (GUI) but also the overall behaviour of an application.

Keywords cross-platform design, agile development, model driven engineering, smalltalk

1. Introduction

Nowadays, general public applications or specific information systems are often able to run on mobile platforms as well as on conventional platforms.

This recent requirement has impacts on the development model, the tools and the development infrastructure. While the cost of purchasing applications decreases, the cost of development increases.

Many research works aim at reducing the development cost. Prototyping is one solution. It can be used to adjust an application before its development for the target platform. In this context, the right capabilities of Smalltalk regarding agile development and prototyping are recognized. Using Smalltak, a prototype can be early tested. However, we are not aware of a Smalltalk infrastructure able to generically deal with different execution platforms for the same application prototype. Moreover, producing the final application to the target platform may involve a significant cost. To overcome this overhead, Model Driven Engineering (MDE) and automatic code generation are often used : stemming from specific models, all or part of the application is automatically generated for the target platform. Produced applications are validated by running them on the target platform. The advantage of this method is to have a single, standard meta-model for all application variants. However, the development cycle is still expensive because the applications have to be tested and validated after code generation.

In this paper, we propose a development method and a framework called Dali based on Smalltalk for early consideration of different execution platforms. With Dali, an application model can be used for several platforms and interpreted before code generation. An execution platform is modelled as a set of constraints over a context. These constraints can affect the presentation of the Graphical User-Interface (GUI) but also the overall behaviour of an application.

The contributions described in this article are :

— we present the Dali framework and an agile development method for early consideration of different execution platforms ;
— we show how an application can be tested before code generation thanks to the interpretation of application models while considering the different execution platforms envisaged for the application.

The remainder of this paper is organized as follows. Chapter 2 explains the problems addressed and existing solutions. Chapter 3 presents our solution called Dali. Chapter 4 presents an illustrative case. Chapter 5 presents Dali meta-models. Chapter 6 explains how Dali is implemented in Pharo. The article ends with relative works and a conclusion.

2. Problems and state of the Art

Our concern is about developing small applications, especially information systems that can be characterized as follows :

1. development is physically centralized on one site ;
2. implementation needs a small number of developers (between 1 and 3) ;
3. targeted system is not critical, i.e. malfunctioning is not likely to endanger people life or health and does not involve significant financial loss to the development company ;
4. the number of screen views is low (between 10 and 20).

These assumptions are in favor of the application of agile methods [TFR05]. Indeed, according to the manifesto for agile software development [BBB+09], some fundamental principles of agile methods are :

— the priority is to meet the needs ; the customer is part of the development team and he operates continuously for validation tasks ;
— to measure the progress of the project regarding the working system and to deliver frequent releases of the system under development ;
— to maintain the simplicity of developments [Hic11].

Considering these principles include notably the following practices [TFR05] :

— development granularity is small ; the code is built iteratively and incrementally and iterations are short ; the expected gain is the early detection and correction of malfunctions ;
— prototypes are developed and tests performed early in the development cycle in relation to the needs expressed by the user ; verification, validation and prototyping activities can be conducted continuously at each iteration.

These practices are consistent with agile development. However, because of the growing importance of mobile platforms, applications shall be developed for desktops and simultaneously adapted to smartphones and tablets.

The idea is similar to that of usage contexts [SeTC99] about the execution platform, the user and the environment. For an application, considering numerous possible contexts involves more complex developments and thus a loss of agility.

2.1 Parallel versions

To support multi-target platforms, for business with limited resources, one direct solution is to maintain different versions, one for each execution platform. Figure 1 depicts such a solution. For each target platform, a part of the application code is written according to the target platform specific library.

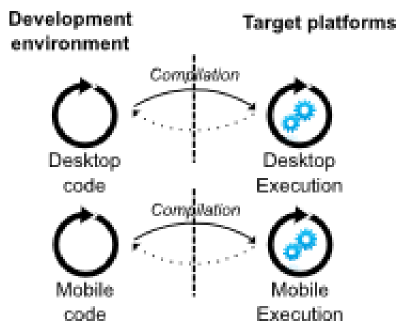


FIGURE 1. Maintaining several versions for the same application

Object-oriented languages can be effectively used to refine abstractions and consider specific contexts such as the features of an execution platform. The solution is to develop a reference version which is then adapted to different platforms envisaged by specialization. Aspect-oriented programming [KLM+97] can provide additional solutions to adapt an application to different contexts.

However, maintaining multiple parallel versions for multiple execution platforms is a very heavy task and makes the continuous validation and tests very difficult. This kind of method can lead to spaghetti code with very negative impact on the evolutivity and on the maintainability of the application code.

2.2 Model-Driven Engineering and code generation

Model-Driven Engineering (MDE) proposes to solve these problems by model transformation and code generation.

As an example, the Model-Driven Architecture (MDA) [Obj00] approach aims at developing a set of models, linked by transformations. These transformations allow to start from a Computation Independent Model (CIM) to a Platform Independent Model (PIM) and finally a final Platform Specific Model (PSM). The PSM represents the concrete implementation of the system. MDE helps to limit specific and non-automated developments for each context : stemming from a cohesive set of models, the different versions can be generated.

The MDE improves the development method because a target platform can be modelled separately. As depicted in Figure 2, from an application model (PIM in MDA) together with a specific platform model, a target platform application model (PSM) is produced by model transformation. Then, parts of the code of the application is automatically generated from it.

The main advantage is that the target platform are specifically modelled and that the same application model can be shared for multiple target platforms.

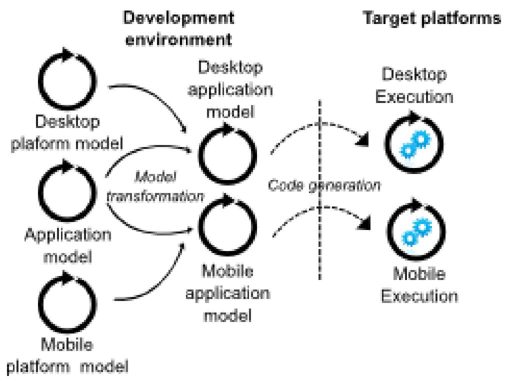


FIGURE 2. The MDE approach

However, depending on the target platform, one must go through a phase of code generation and code compilation before you can execute all or part of the application. These steps are time consuming and the impact of model changes cannot be immediately observed. The causal connection between the models and the generated artefacts is lost because of the code generation step [RFBLO01]. But, to remain agile, a development method must favour short development cycles. Thus, MDE does not help in limiting the loss of agility.

2.3 Emulators

Some development environments provide emulators for target platforms. Emulation relies on the execution of a virtual device supporting the application that will be later deployed. This is, for instance, the solution proposed by the SDK Android [And].

This solution allows to test an application for multiple similar targets without having the physical devices. But, the application execution by the emulator can only occur after the code generation and its compilation. In addition, the application code is specific to one development environment and one kind of emulator. As depicted in Figure 3, this solution implies code generation and deployment steps. In fact, this solution is very similar to the development of parallel versions but with the facility brought by emulators. Because of the code generation step and of the additional deployment steps, the problem of lack of agility is not resolved by emulation. Moreover, emulating can be very slow and time consuming.

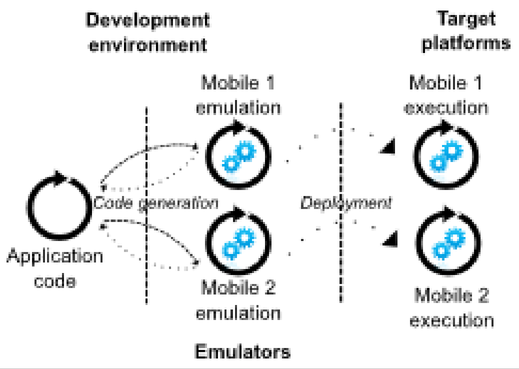


FIGURE 3. Emulation approach

2.4 Common interpreters

A common interpreter can be used for all target execution platforms. Two solutions are possible. The first is depicted by the top part of Figure 4. The interpreter is a separate tool and the application code is the same whatever the platform. This solution is file based. As an example, a web browser can be used as an interpreter of a Javascript program embedded in an HTML page. As another example, one can use Java and its interpreter. The main advantages are that the application code is reusable and web development and deployment is made easier.

Development environments and dynamic languages [Jod10, RG09] promote agile applications development in a uniform and comprehensive way. The second solution is depicted by the bottom part of Figure 4. The interpreter and the development environment are integrated. This is the case of Smalltalk. This solution is image based. As additional advantages, an image based solution favours short development cycle and direct feedbacks. Indeed, with dynamic languages, the causal connection is preserved because any modification of a model can be immediately observed.

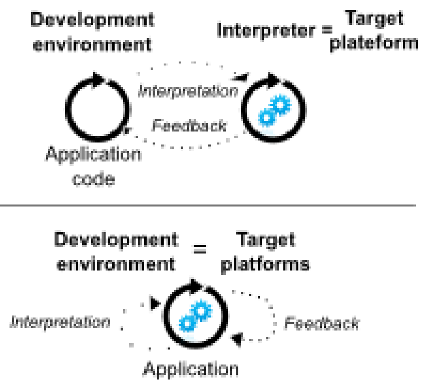


FIGURE 4. Interpretation approach

However, it is necessary to have up-to-date virtual machines for all target platforms. Interpreters can be made available by vendors with specific libraries. This might imply greater difficulties to maintain code reusability. In addition, the use of interpreters can make it difficult the integration of native widgets and more generally using primitive system. Finally, as for emulation, the interpreting process can be time consuming.

3. Dali solution

To improve the development of applications for different execution platforms while allowing us to have native versions, our solution is blended. We propose to use Smalltalk as a tuning and as a development environment and to use MDE to produce target applications for execution platforms.

Maintaining the causal connection between the models and the executed application can resolve significantly the lack of agility. To maintain it, a solution is to execute an application in and by the development environment.

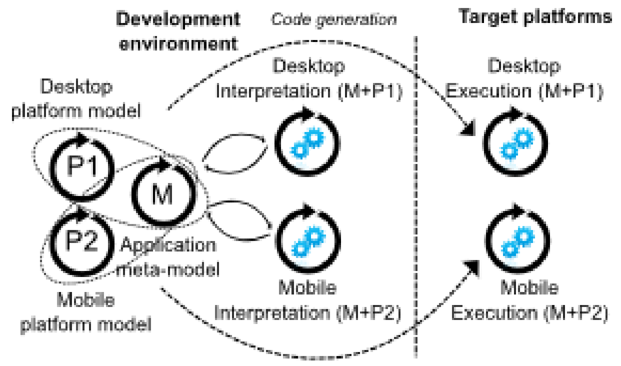


FIGURE 5. The Dali approach

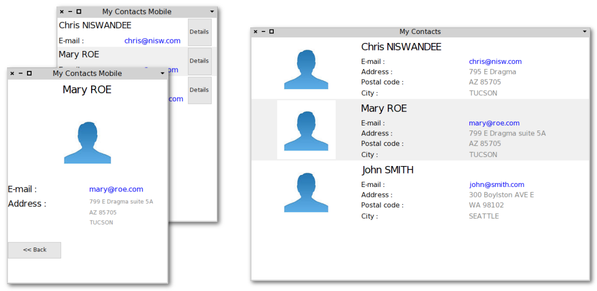
Figure 5 shows the Dali approach. The first goal is to maximize the agility of the development process by the use of a Smalltalk solution to design, prototype and validate an application for several target platform. With Dali, an application model (the business part) must be associated with a specific platform model to be interpreted. The same application model is used for several platforms.

The deployed application can be the Smalltalk one but it might be desirable to directly benefit from native libraries and from fast running. For that purpose, an MDE approach is used. With Dali, after a tuning process, a native application can be automatically generated from the application model together with the chosen target platform model.

The remainder of this paper presents the Dali solution regarding the design on validation steps in Pharo.

4. An illustrative example

Let us take for example the display of a contact list in an address book. Figure 6 shows two possible displays for a contact list. The displayed information as well as the behaviour are different. The left window is adapted to a smartphone and the right window to a desktop computer. On a computer screen, it is possible to view all contact information. On a mobile device, the size is limited. This constraint implies that only the name and the e-mail are presented. A button is added to access the contact details.



Chris NISWANDEE
E-mail : chris@nisw.com
Mary ROE

Mary ROE
E-mail : mary@roe.com
Address : 799 E Dragma suite 5A
AZ 85705
TUCSON

Chris NISWANDEE
E-mail : chris@nisw.com
Address : 795 E Dragma
Postal code : AZ 85705
City : TUCSON

Mary ROE
E-mail : mary@roe.com
Address : 799 E Dragma suite 5A
Postal code : AZ 85705
City : TUCSON

John SMITH
E-mail : john@smith.com
Address : 300 Boylston AVE E
Postal code : WA 98102
City : SEATTLE

FIGURE 6. Two possible displays of a contact list

We do have the same information system but with two different presentations. We also have a behaviour variation because the mobile phone version requires the use of an additional button.

The solution implemented with Dali let us use the same application model and to express the constraints of the mobile phone. Throughout this article, the presentation of the framework will use this example.

5. The Dali framework

To achieve a satisfactory level of agility in considering specific execution platforms, one solution is to allow the developer to model the various platforms and provide an execution of the application within the development environment. The idea is to converge to a suitable solution before code generation.

The remainder of this section presents the meta-model and the main framework components. We depict features related to the description of business models : specific behaviour, presentation and constraints of execution platforms.

5.1 The Dali meta-model

Dali was inspired by self-descriptive object-oriented meta-models such as EMOF [Gro04] and ECore [SBPM09], especially for the description of the structural aspects of an object. In fact, as in EMOF and ECore, Dali is based on a Classifier (or Class) concept, on Operation and on Property definitions. Dali is also inspired from Magritte [RDK07], from its syntax and from its concept of accessors strategy. Moreover, Dali allows the description of the behaviour and of GUI. Specific requirements related to the target platforms can be used to constrain any Dali description element. The business behaviour is expressed in Smalltalk.

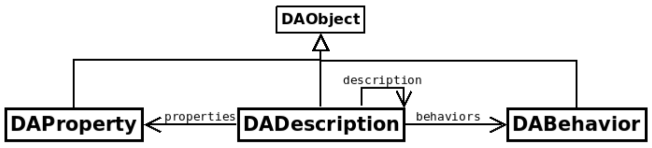


FIGURE 7. Main classes of the Dali meta-model

Figure 7 presents the three main Dali abstractions : *DADescription*, *DAProperty* and *DABehavior*.

A *DADescription* can be related to a Class in EMOF [Gro04]. A *DADescription* handles a set of *DAProperty* and a set of *DABehavior*. A *DADescription* is also a composite : it can contain child descriptions.

A *DAProperty* corresponds to an object instance variable or a Property in EMOF [Gro04]. As a kind of value holder, at runtime, each *DAProperty* stores the value of the related property. A *DAStyle* is a special property used to describe graphic features of a *DAWidget*.

A *DABehavior* might be a business operation (*DAOperation*) as in EMOF [Gro04] but also a more specific behaviour such as a reaction (*DAReaction*), or a binding (*DABinding*).

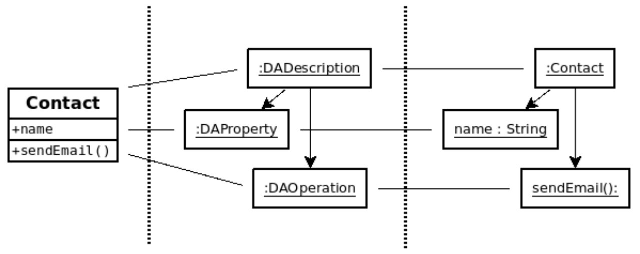


FIGURE 8. Contact class (left), its Dali representation (center) and the simulated instance (right)

Figure 8 shows on the left an UML element for a Contact class with a field $\#name$ and a method $\#sendEmail$. The role of a *DADescription* is twofold : on one side, it describes the structure and the behaviour of the related class, on the other side, it is directly used at runtime to manipulate the value of the properties and to invoke the object behaviour. In the center of Figure 8, is depicted how the Contact class is described with Dali with a *DADescription* instance. The name field is described by a *DAProperty* instance and the $\#sendEmail$ operation is described by a *DAOperation* instance. On the right, is shown a simulated *Contact* instance at runtime. The actual name is stored in the related *DAProperty*. The actual $\#sendEmail$ method is invoked through the related *DAOperation*.

5.2 The behaviour

Dali uses Smalltalk to describe business behaviour. Smalltalk was chosen because the business behaviour can be directly interpreted within the development environment and because it let us the possibility to manipulate the abstract syntax tree (AST) in order to translate the business behaviour to other languages. All business behaviours are manipulated by Dali through instances of *DAOperation*. In fact, a *DAOperation* references the name af the method which implements the related business behaviour.

DABinding and *DAReaction* are more specific. They are implemented as *Observers* [GHJV95] and describe interactions between children of a *DADescription*.

A *DAReaction* specifies an association between a *DAOperation* and an event. For instance, a *DAReaction* can be used to trigger a *DAOperation* as a reaction of an incoming event. Such event can be emitted by a *DAWidget*.

A *DABinding* binds several instances of *DAProperty* together by a source / destination relationship (Data Binding) or bidirectional relationship (Two Way Data Binding). When the source is changed, the destination property is automatically updated (and vice versa if the binding is bidirectional). For instance, a *DABinding* can be used to bind a property of a domain object to a property of a widget.

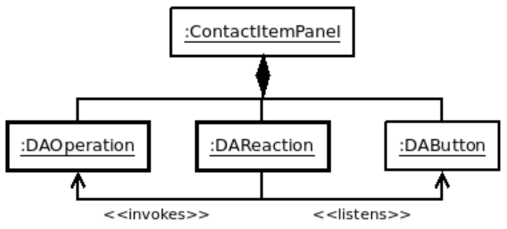


FIGURE 9. Using of a DAReaction

Figure 9 presents an operation triggered as an event reaction. A *DAReaction* listens to an event coming from a source object an is able to trigger an associated operation.

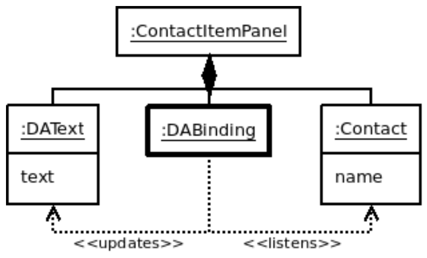


FIGURE 10. Using of a DABinding

Figure 10 depicts how a *DABinding* can be used to automatically chain property updates. When a property value is changed, an event is emitted by the *DAProperty* instance. As a reaction, the *DABinding* updates its target property value.

5.3 The widgets

In Dali, widgets and associated layout are kinds of *DADescription*. Widgets are described as instances of *DAWid-*

get. Each widget has its own set of styles which impacts on the look of the user interface and its own set of managed events which impacts its behaviour. A composition of widgets is specified by a *DALayout* which is owned by a *DAPanel*.

A *DAWidget* is only a logical description which needs to be tied to a native Smalltalk widget at runtime when the application is interpreted. In order to relate a *DAWidget* to a Smalltalk widget, a *DAAdapter* must be specified.

The set of *DAWidget* are defined according to well known standards such as *W3C CSS2* [BLLJ08] for the graphical properties (*DAStyle*) and *W3C DOM* [HKL+13] for events (*DAEvent*). These standards provides the consistency of the API which can cover most of the needs of a GUI description.

5.4 The target platforms modeling

In order to consider execution platforms during the application development, we need models to represent them.

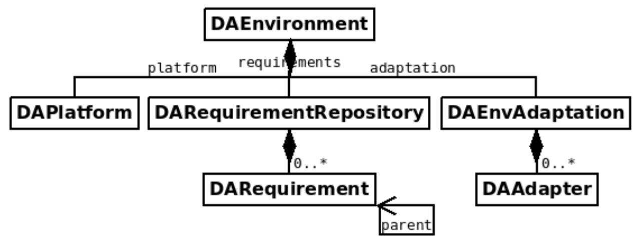


FIGURE 11. The environment composition

As depicted in Figure 11, Dali includes the concept of environment. An environment is reified as a *DAEnvironment* instance. The role of an environment is threefold, it contains the model of the execution platform (*DAPlatform*), the requirements of the application (*DARequirementRepository*) and specifies how the development environment must be customed during the simulation (*DAEnvAdaptation*). Dali makes it possible to simulate and test the same application for several execution platforms. The same set of *DADescription* can be simulated with different actual presentations.

A *DAPlatform* describes a particular target execution platform. A *DARequirementRepository* contains all *DARequirement* used to constrain an application. A *DARequirement* consists in a set of conditions that are checked over a target *DAPlatform*. At runtime, Dali objects structure are set-up according to the related set of *DARequirement*. A *DARequirement* can have a parent requirement. The parent relation is used by Dali to order requirements evaluation, a parent requirement being evaluated before a child one.

Regarding the illustrative example, for the mobile device, a set of *DARequirement* is declared to constrain the size of the panel. Moreover, each row is constraint to present a button instead of showing the complete card data. This kind of constraint is also expressed with a *DARequirement*.

In order to run an application within the development environment, one must declare the actual set of widgets to use (e.g Spec widgets or Morphic widgets). For that purpose, a *DAEnvironment* is set-up with a set of adapters. The purpose of an adapter is to bind a platform widget with a corresponding *DAWidget*. An adapter establishes a mediation between a platform widget and a Dali widget. The mediation consists in providing a concrete look but also in interpreting events and the widget behaviour.

The set of *DAAdapter* is stored within a *DAEnvAdaption* instance. As an example, a *DAEnvAdaption* can be specified for Spec widgets [VRDF12] and another one to bind DAWidget instances directly to Morphic widgets.

6. Dali within Pharo

Dali is implemented in Pharo. The idea is to use Pharo as a classical development environment to design and to early validate applications for multiple target execution platforms but with enhanced agile validation capabilities.

After explanations about the fundamental aspects of Dali regarding DADescription implementation, the reminder of this chapter describes how our illustrative example is implemented in Pharo.

6.1 Fundamentals

All objects manipulated through Dali are instances of a subclass of *DADescription*. Then, a model is made of a set of *DADescription*. It embeds a set of business object descriptions but also the related widget descriptions.

In a Dali description, objects are referenced by a relative identifier (RID). An RID is unique in the scope of a *DADescription* and for all the *DAObject* of a same kind : properties, behaviours and descriptions.

A *DADescription* is made of a list of properties. But, in order to take into account several possible structures and several possible behaviours, the list of properties of a DADescription is not fixed. It depends on the related environment. It means that the actual representation of a concept may differ depending on a given environment.

Each property can be either a data property (e.g. a Contact name) or a behaviour (e.g. the sendMail function). All properties of a *DADescription* are managed as instances of *DAProperty* or of *DABehavior*. It means that a data property is not managed and used through an instance variable but through the corresponding *DAProperty* instance. It means also that a behaviour is not directly coded within a method and, at run-time, is not invoked by a direct message sent but indirectly through a *DABehavior* instanddce.

The methods that implement a particular business object behaviour is still implemented as a method of its class (a subclass of *DADescription*) so that $self$ is preserved at invocation time. Such a method is referenced and indirectly invoked by the corresponding *DAOperation*.

6.2 Configuring an environment

A Dali description structure and behaviour depend on a environment. Thus, an environment must be primarily set-up before instantiating any *DADescription*. The setting of the environment shown in Figure 12 corresponds to the *DAEnvironment* used in the illustrative example.

```
platform := DAPlateform new
        at: #screenWidth put: 300;
        at: #screenHeight put: 400; yourself.

commonRequirement := DARequirement new
        name: #common;
        constraint: [:plfm | true ]; yourself.

smallScreenRequirement := DARequirement new
        name: #smallScreen;
        parent: #common;
        constraint: [:plfm |
                (plfm at: #screenWidth) <= 360 and:
                (plfm at: #screenHeight) <= 480 ]; yourself.

applicationRequirements := DARequirementRepository new
      addRequirement: commonRequirement;
      addRequirement: smallScreenRequirement; yourself.

morphicAdaptation := DAEnvAdaptation new
                adapt: TextMorph
                to: DAText
                with: DAMorphTextAdapter; yourself.

myEnv := DAEnvironment new
    adaptation: morphicAdaptation;
    plateform: platform;
    requirements: applicationRequirements; yourself.

myEnv withinDo: [ContactListWindow new open ].
```

FIGURE 12. Setting and using an environment

An environment associates a *platform*, some *requirements* and an *adaptation* :

— *platform* : A *DAPlateform* stores characteristics of a particular logical platform. All characteristics are stored into a private dictionary so that any characteristic can be freely defined. In Figure 12, an instance of *DAPlateform* is configured to represent a platform with display size characteristics. Regarding our illustrative example, the mobile platform can be described simply by the size of its display.

— *requirements* : A *DARequirement* consists in the specification of a constraint over the platform characteristics. A *DARequirement* is named and is configured with a block. The block implements a constraint over the platform. The constraint block receives the current platform as argument and contains a boolean expression. Finally, all requirements are made available to an environment through a *DARequirementRepository*. Regarding our example :
  — The block of the requirement named $\#common$ always returns $true$, meaning that this requirement is always valid regardless of the platform. Given that the actual list of properties or of behaviours of a *DADescription* is built by evaluating the requirements, this kind of requirement is used to declare that a property or a behaviour is always present in a description.
  — The requirement named $\#smallScreen$ constrains the size of the display. This requirement is used to select the layout to use and which widgets to display.

— *adaptation* : A *DAEnvAdaption* specifies which adapters are used. In our example, we use Morphic as the underlying presentation layer. The adaptation is made of a *DAMorphTextAdapter* which binds *DAText* and *TextMorph*, the Morphic class that is used to edit or display text.

The last line of Figure 12 shows the instantiation of an application configured for a given platform. The environment to be used by the application is made available thanks to the $\#withinDo$: message sent. The application is actually built in the block passed as argument. Internally, the current environment is made available to the application by using the stack context ($thisContext$). Because of the requirement $\#smallScreen$, the resulting application is adapted to a small mobile device as shown in the left part of Figure 6.

6.3 Declaring a business object

The purpose of our illustrative example is to manipulate a list of contacts. A contact is a business object specified by a subclass of *DADescription*. Figure 13 shows the declaration of the *Contact* class.

```
DADescription subclass: #Contact
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'Dali−ContactExample'

Contact>>declareName
  <dali:#common>
  ^DAProperty new
        rid: #name; yourself
```

FIGURE 13. Declaring a business object class

As mentioned in Chapter 6.1, business object properties are not handled through instance variables. Indeed, in Figure 13 the $instanceVariableNames$ string is empty even if a Contact has properties. In fact, any instance variable can be added for internal or private use. But, the business properties must be declared with the help of dedicated methods.

As an example, the Contact *name* property is declared by the $declareName$ method. The property declaration is based on a specific annotation (*pragma* in Pharo). This annotation is used by Dali to identify the methods to evaluate in order to build the actual list of properties. The property selection is achieved according to the constraints that are declared for the current environment. Such a method is a kind of factory method that returns a configured *DAProperty*.

Thus, with the help of its annotation, the *declareName* method declares that the property named *#name* is added if the requirement named *#common* is met. As shown in Figure 12, the *#common* requirement is always met because its constraint block returns *true*. As a consequence, the *#name* property is always added to the *Contact* description whatever the environment.

```
Contact>>declareLocation
   <dali:#( #smallScreen #gps ) >
   ^DAProperty new
         rid: #location; yourself
```

FIGURE 14. Declaring a property with multiple requirements

In some cases, using a single requirement in the annotation might lead to duplicate requirement code. For instance, suppose that we want to use the GPS service on a smartphone. A field named *location* would be required to store the GPS data. As a consequence, for this property, two requirements must be met : the application runs on the small screen device and the GPS service is available. Figure 14 shows how multiple requirements can be declared with the annotation.

At runtime, a property is used as a value holder. In order to get or set the value of a property, accessors must be implemented. Regarding the *#name* property, we might declare its accessors as shown in Figure 15. Notice that the name of the method is meaningful because it is considered by Dali as the name of the corresponding property. The property instance retrieval is achieved by the *thisProperty* message sent.

```
Contact>>name
   ^self thisProperty value

Contact>>name: aString
   self thisProperty value: aString
```

FIGURE 15. Declaring a property accessors

6.4 Declaring a widget

As explained in Chapter 6, a *DAWidget* is also a *DADescription*. As a consequence, declaring a widget property and its accessors is achieved with the same syntax as for the business objects. A widget is specified through a subclass of *DAWidget*.

Figure 16 shows the declaration of the class *ContactItemPanel* that is the description of the presentation of a contact in the list of contacts shown in Figure 6. A *ContactItemPanel* is specified by a subclass of *DAPanel* which can handle sub-widgets embedded in a layout.

```
DAPanel subclass: #ContactItemPanel
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'Dali-ContactExample'

ContactItemPanel>>declareContact
   <dali:#common>
   ^DAProperty new
         rid:#contact; yourself

ContactItemPanel>>declareNameText
   <dali:#common>
   ^DAText new
         rid:#nameText;
         fontSize: 22 pt; yourself
```

FIGURE 16. Declaring a widget class

As for a property, a sub-widget can be declared by a dedicated method. As shown in Figure 16, the *DAText* named *nameText* is declared by the *declareNameText* method. This method is also used to configure the property *fontSize* of the *DAText* instance.

6.5 Adding a layout

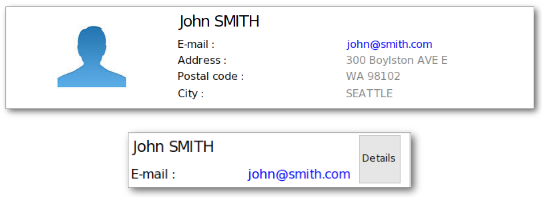
Figure 17 shows the declaration of a row layout referencing widgets of the *ContactItemPanel*. The way a layout is declared with Dali is very near from Spec [VRDF12]. A layout consists simply in a tree of sub-widget references that can be visited without any adaptation and drawing.

```
ContactItemPanel>>declareLayout
      <dali:#common>
      ^DARow new
         child:#contactImage;
         column: [ :c |
               c child: #nameText;
                  row: [ :r |
                        r child: #emailLabel;
                           child: #emailText ]]; yourself
```

FIGURE 17. Adding a layout

6.6 Specifying multiple representations

As a solution to multiple representations of the same business object, a Dali description contains all the possible properties regardless of the actual environment. When a description is bound to an environment, the subset of properties that meet the environment requirements is computed.


John SMITH
E-mail : john@smith.com
Address : 300 Boylston AVE E
Postal code : WA 98102
City : SEATTLE

John SMITH
E-mail : john@smith.com
Details

FIGURE 18. Focus on the presentation of a contact

Figure 18 shows the same element of a contact list but with two different presentations. The top one is for desktop whereas the bottom one is for a mobile platform. This difference is implemented with two possible layouts in the *ContactItemPanel* class. The desktop version is shown in Figure 17. For the mobile version shown in Figure 19, one must declare an additional button and use the *dali* annotation to bind the declarations with the $\#smallSreen$ requirement.

```
ContactItemPanel>>declareSmallLayout
  <dali:#smallScreen>
  ∧DARow new
        child:#nameText;
        child:#detailsButton; yourself

ContactItemPanel>>declareDetailsButton
  <dali:#smallScreen>
    ∧DAButton new
        rid:#detailsButton;
        text: 'Details'; yourself
```

FIGURE 19. Adapting widget with requirements

At instantiation time, the two applicant layouts may be selected. Indeed, the $\#common$ requirement is always met whatever the platform. Dali resolves this selection issue by using the parent relation between requirements. As shown in Figure 12, the $\#smallScreen$ requirement is a child of the $\#common$ requirement. As a consequence, it is always evaluated after the $\#common$ requirement and then, in the context of a mobile platform, only the mobile layout is selected and actually instantiated.

6.7 Adding behaviours

As introduced in Chapter 5.2, a behaviour can be either a *DAOperation* or a *DABinding* or a *DAReaction*. The same declaration syntax is used as for properties, widgets and layouts.

6.7.1 Adding an operation

An operation consists in the declaration of a method to invoke. The name of the method serves as a key to lookup the actual method. Figure 20 shows the declaration of a *DAOperation* and the associated method. At runtime, this operation invocation results in the $\#openDetails$ message sent to the receiver.

```
ContactItemPanel>>declareOpenDetails
  <dali:#common>
  ∧DAOperation new
        rid:#openDetails; yourself

ContactItemPanel>>openDetails
    ContactDetailsWindow new
        contact: self contact;
        open
```

FIGURE 20. Declaring an operation

6.7.2 Adding a binding

Figure 21 shows the declaration of a *DABinding*. The *DABinding* establishes a link between the property $\#name$ of the Contact and the property $\#text$ of the child widget $\#nameText$. Thanks to this binding, when the contact name is changed, the $\#text$ property of the widget is automatically updated.

```
ContactItemPanel>>declareNameBinding
  <dali:#common>
  ∧DABinding new
        rid:#nameBinding;
        srcAccessor:
            ((#contact asDaliAccessor)
                withNext: #name asDaliAccessor);
        destAccessor:
            ( (#nameText asDaliAccessor)
                withNext: #text asDaliAccessor);
        yourself
```

FIGURE 21. Declaring a binding

A binding relies on accessors for the source and the target properties. Figure 22 shows the hierarchy of accessors implemented as a Decorator. An accessor can be straightforward or chained. This feature is borrowed from Magritte [Ren06].

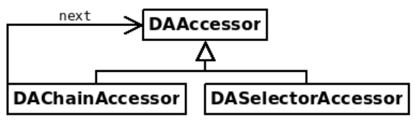


FIGURE 22. Hierarchy of accessors in Dali

6.7.3 Adding a reaction

A reaction consists in the declaration of an association between an event and an operation. Figure 23 shows the declaration of a *DAReaction* which listens the event *DAClick* fired by the button named $\#detailsButton$. When the button is clicked, the operation named $\#openDetails$ is invoked.

```
ContactItemPanel>>declareButtonReaction
    <dali:#smallScreen>
    ^DAReaction new
            rid:#buttonReaction;
            event: DAClick;
            senderAccessor: #detailsButton asDaliAccessor;
            operationRid: #openDetails; yourself
```

FIGURE 23. Declaring a reaction

6.8 More on styles and events

The public protocol of a widget consists mainly in implementing its styles and managing events.

Dali is a modelling layer that must be adapted to specific environments for graphic rendering. To remain adaptable to any underlying framework, Dali does not provide its own underlying graphical implementation and widget protocols are not defined according to a specific one (Morphic, Spec,...). Instead, the goal is to rely on normalized protocols. For that purpose, we have chosen to be compliant with two main standards : W3C CSS2 [BLLJ08] for styles and W3C DOM [HKL+13] for events.

6.8.1 Styles

Styles management can be confusing. Each style can be used by several widgets but sharing of styles can not rely on widget hierarchy definition. As an example, button and text are represented in Dali by *DAButton* and *DAText*. Their common ancestor is *DAWidget*. These two widget kinds can manage a text. Then, these widget kinds must implement text styles management. Unfortunately, the text styles management is not available at the level of *DAWidget*. Thus, we need a clear way to define styles and to allow their sharing independently of the widget hierarchy. For that purpose, as shown in Figure 24, Dali uses traits to declare available styles.

```
DAWidget subclass: #DARectangle
    uses: DATWithBackground + DATWithBorder
    ...
DARectangle subclass: #DAButton
    uses: DATWithText
    ...
DAWidget subclass: #DAText
    uses: DATWithText + DATWithBgColor
    ...
```

FIGURE 24. The text and the button widgets sharing the same text style definition

A style trait covers the definition of a standard subset of CSS2 [BLLJ08] styles. As shown in Figure 25, in such a trait, each style implementation is made of three methods : the first is for the style instantiation and the two others are for the style value accessing. As an example, The *DATWithText* trait shown in Figure 25 is compliant with subset of CSS about text specification.

```
DATWithText>>declareFontStyle
        <dali>
        ^DAStyle new rid: #fontStyle;
              value: DAFontStyle normal; yourself

DATWithText>>fontStyle
        ^self thisProperty value

DATWithText>>fontStyle: aFontStyle
        ^self thisProperty value: aFontStyle
```

FIGURE 25. DATWithFont trait snippet

6.8.2 Events

The DOM [HKL+13] specification define different kind of events that can be fired by graphical elements. Dali provide the *DAWidgetEvent* hierarchy which is compliant with this specification. To take advantage of announcement mechanism in Pharo, *DAWidgetEvent* is a subclass of *Announcement*. Once more, traits are used to make available the event protocols at the widget level. The implementation of an event, in such a trait, is made of two declarations of a *DAOperation* with their related methods. The first method fires a specific event, the second method declare a reaction between this event and an operation. Figure 26 shows the first one.

```
DATWithClickEvent>>declareFireClickEvent
        <dali>
        ^DAOperation new rid:#fireClickEvent; yourself

DATWithClickEvent>>fireClickEvent
        self announcer announce: (DAClick target: self)
```

FIGURE 26. DATWithClickEvent trait snippet

6.9 Visiting a model

As a development environment, Dali permits early validation of applications through their direct interpretation. But the ultimate goal is to produce an adapted version of an application for a specific execution platform. The primary requirement for such a goal is to be able to visit a description. Moreover, a description model must be visitable even it is not adapted.

To visit a model instance, an environment must be configured as described in Chapter 6.2. This environment represents the target platform. In its current state, Dali allows such visits as presented in Figure 27. This example shows a visit invocation in order to generate a CSS representation. The resulting CSS is shown in Figure 28.

```
myEnv whithinDo: [
        DACssExporter new
           visit: (ContactListWindow new)]
```

FIGURE 27. Exporting CSS from a widget

```
div.contact−window {
      width: 300 px; height: 400 px;
      color: #000000; background−color: #FFFFFF;
}
 div.contact−item−panel{
      width: 100%; height: 60 px;
      color: #000000; background−color: #FFFFFF;
}
span.contact−item−panel .contact−name { font−size : 22 pt;}
span.contact−item−panel .contact−email { color : #0000F1;}
```

FIGURE 28. Generated CSS snippet

7. Related Work

AppliDE [QDDD11] is a software framework that is based on Software Product Lines (SPL) [PBL05]. SPL lets the user define characteristics diagrams for each products family also called features. AppliDE uses SPL in association with MDE to construct a single model of products family. Features are used to specify the variability between different products. This model is used to automate the derivation of applications for several target platforms. AppliDE describes the behaviour by high-level services (GPS, mailing,...) and manipulates them as features. The business behaviour is implemented at the target execution platform level. The generated code is linked with the business behaviour after code generation. AppliDE uses a static builder for GUI aspects of an application. AppliDE is code generation based. It does not provide any interpreting facility.

CAPucine [Par11] uses also SPLs but with Aspect-oriented programming to consider context variations at runtime. It is based on a variability model to define product families and their variability points. An aspect model is used at design time to generate code and is used at runtime for reconfigure an application according to the events fired within the execution context. CAPucine is also based on code generation. The behaviour is represented at a high abstraction level, as components or services. Compared to AppliDE, CAPucine does not include any elements to automate the development of the GUI.

SPL features are similar to the Dali notion of requirements. But, the composition of an application is made at high level of abstraction (service level). Whereas, in Dali, a composition of an application is made at the class level.

The works around the plasticity [Cal10] and the multimodality [CCT+03] of GUI take advantage of MDA [Obj00] architecture to abstract and generalize concepts without specialize an application directly to some context. But to allows reconfiguration at runtime, the target platforms must include a specific framework. The Comets [CDCD05] approach brings widgets which can react at runtime to adapt their presentation according to the variation of the runtime context. These Comets are implemented in each target platform and their models are used at design time to specify GUI composition.

In Dali, a platform is modelled at design time and is made to constrain objects structure. But, Dali does not provide any dynamic reconfiguration mechanism after code generation. Thus, the meta-model is not represented at runtime in the target platform.

Magritte [RDK07] is a framework to describe properties of a domain object. The properties description consist in a separate meta-model that can be manipulated apart. It is auto-descriptive. Magritte uses mementos to cache model state and manage model changes as transactions. These mementos can be also used as a kind of value holder.

Dali uses also a meta-model to describe object properties. In Dali, a description is directly used as a business object : its properties serve as value holders and for the invocation of the behaviour.

Spec [VRDF12] is inspired by the VisualWorks UI builder and is implemented in the Pharo environment. It is used to specify and build GUI in Pharo. Spec uses a logical widget model with a specific API and an adaptation layer to bind its widget model to Morphic widgets. The widget composition is implemented with layouts relying on the command pattern. Each layout element exists as a command that is evaluated when the layout is adapted to Morphic. No visiting mechanism is provided for the widget hierarchy.

A part of Dali is dedicated to GUI description. As in Spec, it implements widget composition, layouts and an adaptation layer. Dali widgets are auto-descriptive and can be reconfigured at design time according to platform constraints. Moreover, Dali style and events are compliant with W3C [BLLJ08, HKL+13].

8. Conclusion and further work

This paper deals with the problem of agile application development for multiple execution platforms. Nowadays, application vendors must provide multi-platform products. Often, existing desktop applications have to be adapted or rewritten to be compliant with mobile platform. This constraint impacts the cost of application development even with agile technologies.

This paper presents a solution named Dali which benefits from Smalltalk together with the Model Driven Engineering to allow multi-platform application design. With Dali, an application can dynamically take into account a platform description and can be interpreted to allow agile design and validation. When the application is mature enough, the idea is to finally generate a native target application.

In its current state, Dali provides a framework that can be used to design desktop as well as mobile simple applications.

The current set of available widgets and related adapters remain to be enriched. A Dali model can be visited. It provides the mean to actually generate target platform code. But, generating a real application remains to be experimented.

At the design level, two perspectives are under consideration. The first one concerns the property definitions that could benefit from Slots [VBLN11]. Indeed, a property description is a specific instance variable with particular management rules.

In Dali, target platform description consists in a set of properties. A pragma based mechanism is used to select the actual set of business object properties according to specific environment requirements. The pragma mechanism implies a class based definition of all possible properties. This part of the framework could benefit from an Aspect oriented mechanism. Indeed, the definition of properties available in a given environment could be designed with the help of Aspects. We plan to explore the use of PHANtom [Fab12] for that purpose.

Références

[And] Android sdk. http ://developer.android.com/.

[BBB+09] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, and Jeff Sutherland. Manifesto for agile software development. http ://agilemanifesto.org/, 2009.

[BLLJ08] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2 (CSS2) specification. W3C recommendation, W3C, April 2008. http ://www.w3.org/TR/2008/REC-CSS2-20080411/.

[Cal10] Gaelle Calvary. Plasticité des interfaces homme-machine : Rétrospective et perspectives. pages 3–4, Marseille, France, 2010.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS*, 15 :289–308, 2003.

[CDCD05] Gaëlle Calvary, O. Daassi, Joëlle Coutaz, and A. Demeure. Des widgets aux comets pour la plasticité des systèmes interactifs. *Revue d'Interaction Homme-Machine (RIHM)*, Volume 6, n°1, ISSN 1289-2963 :33–53, 2005.

[Fab12] Johan Fabry. Phantom : An aspect language for pharo smalltalk. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, pages 31–32, New York, NY, USA, 2012. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gro04] Object Management Group. Meta object facility (mof) 2.0 core final adopted specification. 2004.

[Hic11] Rich Hickey. Simple made easy. StrangeLoop 2011 conference talk, http ://www.infoq.com/presentations/Simple-Made-Easy/, Oct 2011.

[HKL+13] Philippe Le Hégaret, Gary Kacmarcik, Travis Leithead, Tom Pixley, Björn Höhrmann, Doug Schepers, and Jacob Rossi. Document object model (DOM) level 3 events specification. W3C working draft, W3C, November 2013. http ://www.w3.org/TR/2013/WD-DOM-Level-3-Events-20131105/.

[Jod10] André Jodoin. *Un environnement dynamique de développement (EDD) pour le prototypage rapide d'interfaces graphiques*. PhD thesis, Université de Montréal, 2010.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Proceedings of ECOOP '97, LNCS, Springer-Verlag*, 1241 :220—-242, June 1997.

[Obj00] Object Management Group. Model Driven Architecture (MDA), 2000.

[Par11] Carlos Parra. *Towards Dynamic Software Product Lines : Unifying Design and Runtime Adaptations*. These, Université des Sciences et Technologie de Lille - Lille I, March 2011.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[QDDD11] Clément Quinton, Christophe Demarey, Nicolas Dolet, and Laurence Duchien. AppliDE : modélisation et génération d'applications pour smartphones. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM'11)*, pages 41–45, Lille, France, June 2011.

[RDK07] Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte - a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2007.

[Ren06] Lukas Renggli. Magritte — meta-described web application development. Master's thesis, University of Bern, jun 2006.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The architecture of a uml virtual machine. *SIGPLAN Not.*, 36(11) :327–341, October 2001.

[RG09] Lukas Renggli and Tudor Gîrba. Why smalltalk wins the host languages shootout. In *IN : PRO-*

CEEDINGS OF INTERNATIONAL WORKSHOP ON SMALLTALK TECHNOLOGIES (IWST 2009), ACM DIGITAL LIBRARY, 2009.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[SeTC99] Angela Sasse, Chris Johnson (editors, David Thevenin, and Joelle Coutaz. Plasticity of user interfaces : Framework and research agenda, 1999.

[TFR05] Daniel E. Turk, Robert B. France, and Bernhard Rumpe. Assumptions underlying agile software-development processes. *J. Database Manag.*, 16(4) :62–87, 2005.

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible Object Layouts : enabling lightweight language extensions by intercepting slot access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, Portland, États-Unis, November 2011.

[VRDF12] Benjamin Van Ryseghem, Stéphane Ducasse, and Johan Fabry. Spec : A Framework for the Specification and Reuse of UIs and their Models. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2012)*, Gent, Belgique, August 2012.