Model Driven Context Aware Reactive Applications

Tony Clark (with Dean Kramer and Samia Oussena, UWL)

School of Engineering and Information Sciences, Middlesex University

October 5, 2011

Contents

Context Aware Reactive Applications

> *Context aware applications are: intelligent applications that can monitor the user's context and, in case of changes in this context, consequently adapt their behaviour in order to satisfy the user's current needs or anticipate the user's intentions.*

L.M. Daniele, E. Silva, L.F. Pires, and M. Sinderen. A SOA-based Platform-specfic Framework for Context-Aware Mobile Applications. Enterprise Interoperability, 2009.

> *Reactive applications must deal with events that are received from their environment and react accordingly.*

D. Harel and A. Pnueli. On the development of reactive systems. Weizmann Institute of Science, Dept. of Computer Science, 1985.

The Rise of CARA

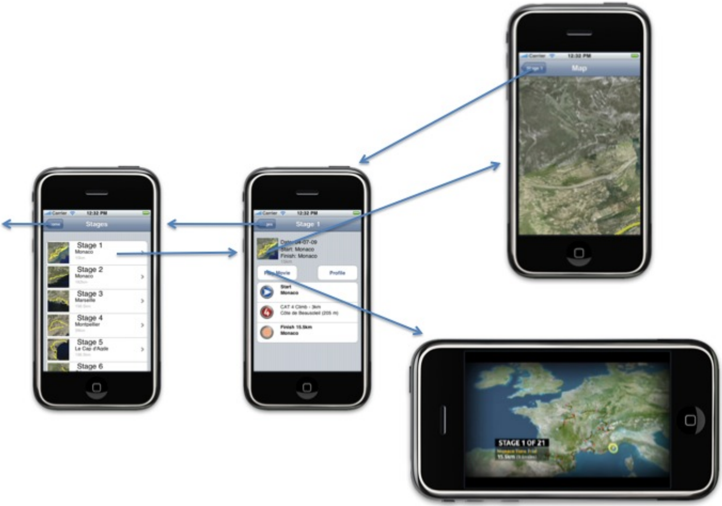
The Rise of CARA

The Rise of CARA

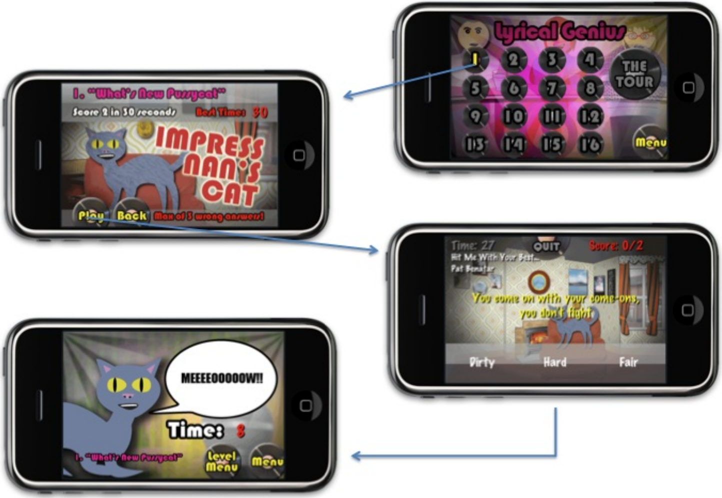
Model Driven Development

- Use models as the primary system artifact.
- Focus on key aspects of systems.
- Use standards to support interoperability and tools.
- Use meta-modelling to construct domain-specific abstractions.
- Analyse, verify and validate the models.
- Generate (parts of) the system from models.

Knowledge Connext and KTP



Knowledge Connext and KTP



Problem and Proposal

- MDD should be available for CARA applications - right?

Problem and Proposal

- MDD should be available for CARA applications - right?
- Some attempts:

Problem and Proposal

- MDD should be available for CARA applications - right?
- Some attempts:
  - Complex or tool specific.

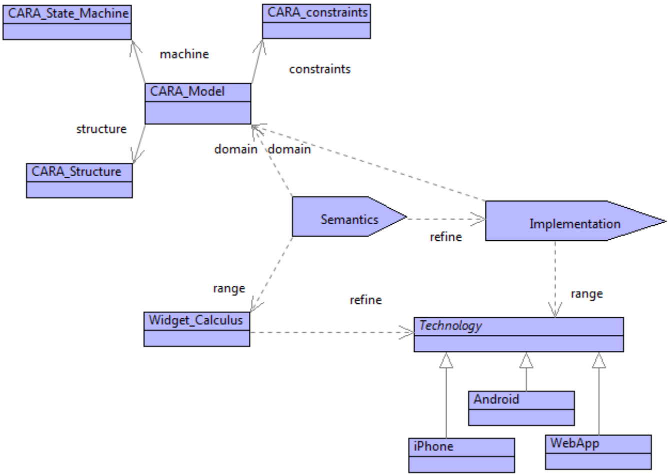
Problem and Proposal

- MDD should be available for CARA applications - right?
- Some attempts:
  - Complex or tool specific.
  - Incomplete, lacks execution.

Problem and Proposal

- MDD should be available for CARA applications - right?
- Some attempts:
    - Complex or tool specific.
    - Incomplete, lacks execution.
- Can we design a simple, universal MDD solution?

CARA Overview



Buddy

- Mobile phones have address lists.

L. Daniele, L. Ferreira Pires, and M. van Sinderen. An MDA-based Approach for Behaviour Modelling of Context-aware Mobile Applications. In Model Driven Architecture-Foundations and Applications. Springer, 2009.

Buddy

- Mobile phones have address lists.
- A phone is always in contact with its network provider.

L. Daniele, L. Ferreira Pires, and M. van Sinderen. An MDA-based Approach for Behaviour Modelling of Context-aware Mobile Applications. In Model Driven Architecture-Foundations and Applications. Springer, 2009.

Buddy

- Mobile phones have address lists.
- A phone is always in contact with its network provider.
- Each phone has a unique address, e.g. `tony@widget.org`.

L. Daniele, L. Ferreira Pires, and M. van Sinderen. An MDA-based Approach for Behaviour Modelling of Context-aware Mobile Applications. In Model Driven Architecture-Foundations and Applications. Springer, 2009.

Buddy

- Mobile phones have address lists.
- A phone is always in contact with its network provider.
- Each phone has a unique address, e.g. `tony@widget.org`.
- Each phone maintains a database of contacts.

L. Daniele, L. Ferreira Pires, and M. van Sinderen. An MDA-based Approach for Behaviour Modelling of Context-aware Mobile Applications. In Model Driven Architecture-Foundations and Applications. Springer, 2009.

Buddy

- Mobile phones have address lists.
- A phone is always in contact with its network provider.
- Each phone has a unique address, e.g. `tony@widget.org`.
- Each phone maintains a database of contacts.
- Users want to know about contacts in their database that are co-located.

L. Daniele, L. Ferreira Pires, and M. van Sinderen. An MDA-based Approach for Behaviour Modelling of Context-aware Mobile Applications. In Model Driven Architecture-Foundations and Applications. Springer, 2009.

Buddy

- Mobile phones have address lists.
- A phone is always in contact with its network provider.
- Each phone has a unique address, e.g. `tony@widget.org`.
- Each phone maintains a database of contacts.
- Users want to know about contacts in their database that are co-located.
- When one phone moves into the vicinity of the other then both phones are told of the availability of the other in terms of the contact address.

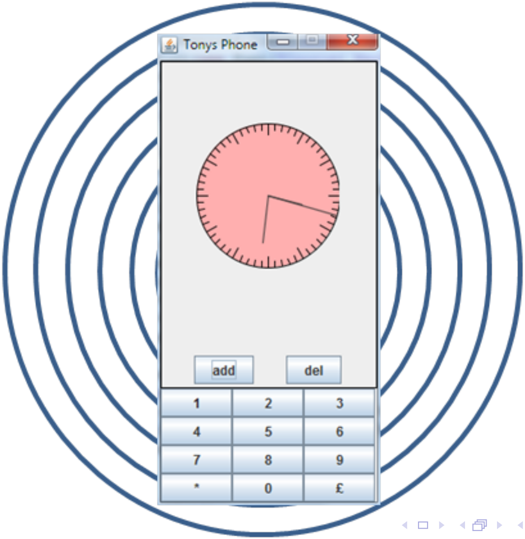
L. Daniele, L. Ferreira Pires, and M. van Sinderen. An MDA-based Approach for Behaviour Modelling of Context-aware Mobile Applications. In Model Driven Architecture-Foundations and Applications. Springer, 2009.

Buddy

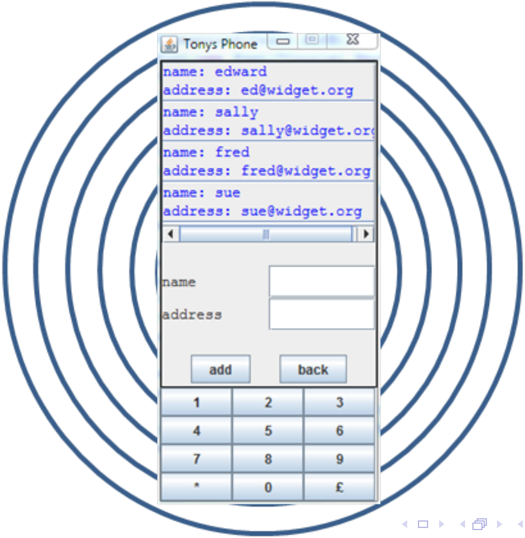
- Mobile phones have address lists.
- A phone is always in contact with its network provider.
- Each phone has a unique address, e.g. `tony@widget.org`.
- Each phone maintains a database of contacts.
- Users want to know about contacts in their database that are co-located.
- When one phone moves into the vicinity of the other then both phones are told of the availability of the other in terms of the contact address.
- If the address is in the user's database then the phone flashes the contact.

L. Daniele, L. Ferreira Pires, and M. van Sinderen. An MDA-based Approach for Behaviour Modelling of Context-aware Mobile Applications. In Model Driven Architecture-Foundations and Applications. Springer, 2009.

Tony’s Phone



Tony Knows Sally

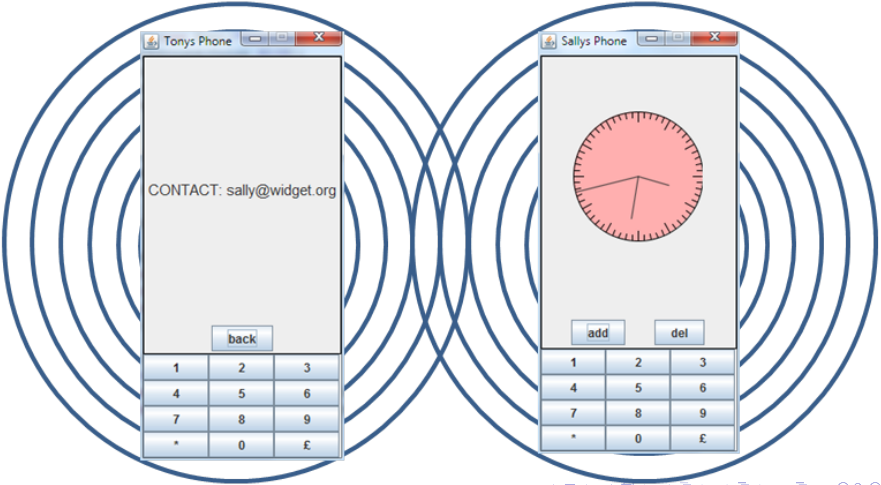


```
name: edward
address: ed@widget.org
name: sally
address: sally@widget.or
name: fred
address: fred@widget.org
name: sue
address: sue@widget.org
```

name □

address □

Context Aware



CONTACT: sally@widget.org

Contents

1. CARA Applications
   - Definitions
   - Research
   - Case Study
2. Modelling
   - Diagrams
   - Widget Calculus
3. Case Study Model
4. Other Issues
   - Types
   - Implementation

Domain Analysis

GUI Trees GUI elements are organized in trees.

Events Elements generate events and listeners handle them.

Platform Each platform provides a different collection of event generators.

Object-Orientation Implementation technologies are mostly OO.

Transitional Behaviour Application execute by performing state transitions.

Data Persistence Applications manage simple databases.

Static Typing Generally lacked by most implementation technologies for platforms.

UML Profile

`<<external>>` Application components supplied by platform.

`<<widget>>` User defined components, tree-shaped.

Containership Associations, black-diamond.

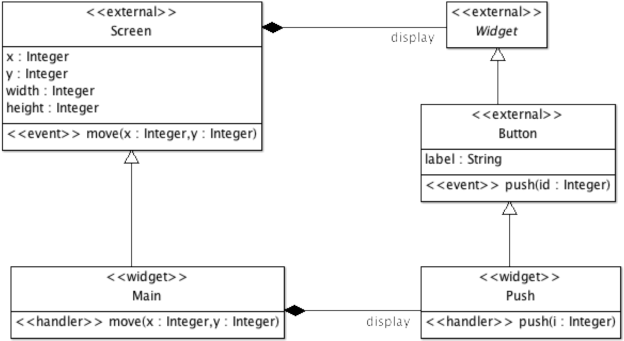
Root Root containers define application states.

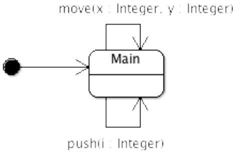
`<<event>>` Generated by the platform, context events.

`<<handler>>` Process events.

Machines State transitions.

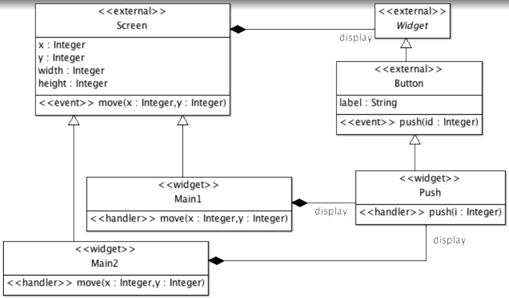
Single Button Model

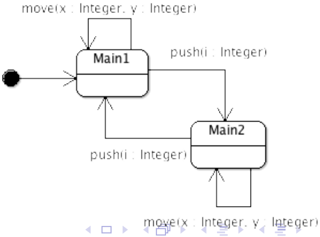




```
context Push inv:
  label = 'PUSHME'
```

Toggle Button Model




move(x : Integer, y : Integer)
Main1
push(i : Integer)
Main2
push(i : Integer)
move(x : Integer, y : Integer)

```
context Main1 inv:
  display.label = 'PUSHME'

context Main2 inv:
  display.label = 'PUSHED'
```

Widget Calculus: Key Features

Take a standard λ-calculus and add:

- widgets: tree-structured records that generate and handle events.
- externals: each platform provides some built-in widgets.
- commands: change the state of the world.
- methods: each external provides some built-in commands.
- cycle: eval, command, wait

CARA models translate onto the widget-calculus. The calculus provides the action-language for the models.

A Button (1)

events	ids	eval

0<-push(0)		main =
		widget self (screen(50,50,50,50,push)) {
		move(x,y) = **do** { **return** self }
		};
		push =
		widget self (button('PUSHME')) {
		push(i) = **do** { **return** self }
		}

A Button (2)

events	ids	command

`0<-push(0)`		`<widget self (screen(50,50,50,50,`
		`  widget self (button(’PUSHME’)) {`
		`    push(i) = do { return self }`
		`  }) {`
		`  move(x,y) = do { return self }`
		`}>`

A Button (3)

events	ids	wait

0<-push(0)	0 1
	2 3
		widget(3) self (screen(2,50,50,50,50,
		widget(1) self (button(0,'PUSHME')) {
		push(i) = **do** { **return** self }
		}) {
		move(x,y) = **do** { **return** self }
		}

A Button (4)

events	ids	eval
		`widget(3) self (screen(2,50,50,50,50,`
		`  <widget(1) self (button(0,'PUSHME')) {`
		`    push(i) = do { return self }`
	0 1	`  }.push(0)>) {`
	2 3	`  move(x,y) = do { return self }`
		`}`

A Button (5)

events	ids	command
		widget(3) self (screen(2,50,50,50,50,
		<**do** {
		return **widget**(1) self (button(0,'PUSHME'))
		push(i) = **do** { **return** self }
	0 1	}
	2 3	}>) {
		move(x,y) = **do** { **return** self }
		}

A Button (6)

events	ids	wait

	0 1
	2 3
		widget(3) self (screen(2,50,50,50,50,
		  widget(1) self (button(0,'PUSHME')) {
		    push(i) = **do** { **return** self }
		  }) {
		  move(x,y) = **do** { **return** self }
		}

Toggle (1)

```
events       | ids | eval
             |     |
             |     | main =
             |     |   widget self (screen(50,50,50,50,push)) {
             |     |     move(x,y) = do { return self }
             |     |   };
             |     | push =
0<-push(0)   |     |   widget (button('PUSHME')) {
1<-push(1)   |     |     push(i) = do { return pushed }
             |     |   };
             |     | pushed =
             |     |   widget (button('PUSHED')) {
             |     |     push(i) = do { return push }
             |     |   }
```

Toggle (2)

events	ids	command
		`<widget self (screen(50,50,50,50,`
		`  widget (button('PUSHME')) {`
		`    push(i) = do {`
		`      return widget (button('PUSHED')) {`
0<-push(0)		`        push(i) = do { return push }`
1<-push(1)		`      }`
		`    }`
		`  })) {`
		`  move(x,y) = do { return self }`
		`}>`

Toggle (3)

events	ids	wait
0<-push(0)	0	
1<-push(1)	1	
	2	
	3	
	4	
	5	

```
widget(5) self (screen(4,50,50,50,50,
  widget(3) (button(0,'PUSHME')) {
    push(i) = do {
      return widget(2) (button(1,'PUSHED')) {
        push(i) = do { return push }
      }
    }
  })) {
  move(x,y) = do { return self }
}
```

Toggle (4)

events	ids	eval
1<-push(1)	0	
	1	
	2	
	3	
	4	
	5	

```
widget(5) self (screen(4,50,50,50,50,
  <widget(3) (button(0,'PUSHME')) {
    push(i) = do {
      return widget(2) (button(1,'PUSHED')) {
        push(i) = do { return push }
      }
    }
  }.push(0)>)) {
  move(x,y) = do { return self }
}
```

Toggle (5)

events	ids	command
1<-push(1)	0	
	1	
	2	
	3	
	4	
	5	

```
widget(5) self (screen(4,50,50,50,50,
  <do {
    return widget(2) (button(1,'PUSHED')) {
      push(i) = do { return push }
  }>)) {
  move(x,y) = do { return self }
}
```

Toggle (6)

events	ids	wait

1<-push(1)	0	`widget(5) self (screen(4,50,50,50,50,`
		`  widget(2) (button(1,'PUSHED')) {`
		`    push(i) = do { return push }`
		`  })) {`
		`  move(x,y) = do { return self }`
		`}`

	1	
	2	
	3	
	4	
	5	

Toggle (7)

events	ids	eval
	0	
	1	
	2	
	3	
	4	
	5	

```
widget(5) self (screen(4,50,50,50,50,
  <widget(2) (button(1,'PUSHED')) {
    push(i) = do { return push }
  }.push(1)>)) {
  move(x,y) = do { return self }
}
```

Toggle (8)

events	ids	command

	0
	1
	2
	3
	4
	5
		`widget(5) self (screen(4,50,50,50,50,`
		`  <do { return push }>)) {`
		`  move(x,y) = do { return self }`
		`}`

Toggle (9)

events	ids	wait
	0 1 2 3 4 5	

```
widget(5) self (screen(4,50,50,50,50,
  widget(3) (button(0,’PUSHME’)) {
    push(i) = do {
      return widget(2) (button(1,’PUSHED’)) {
        push(i) = do { return push }
      }
    }
  })) {
  move(x,y) = do { return self }
}
```

Commands

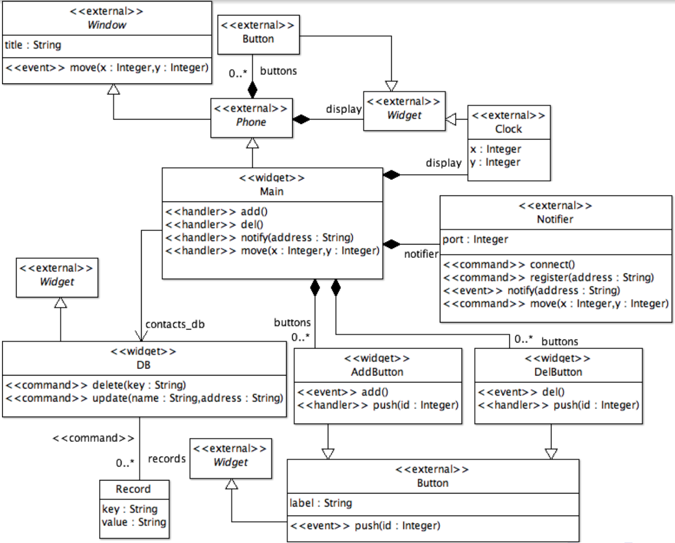
Commands can:

- Create widgets.
- Create, access or update variables.
- Read or write files and databases.
- Access local context, e.g. orientation, battery charge, light levels.
- Connect to servers.
- Write to servers.
- Send emails.

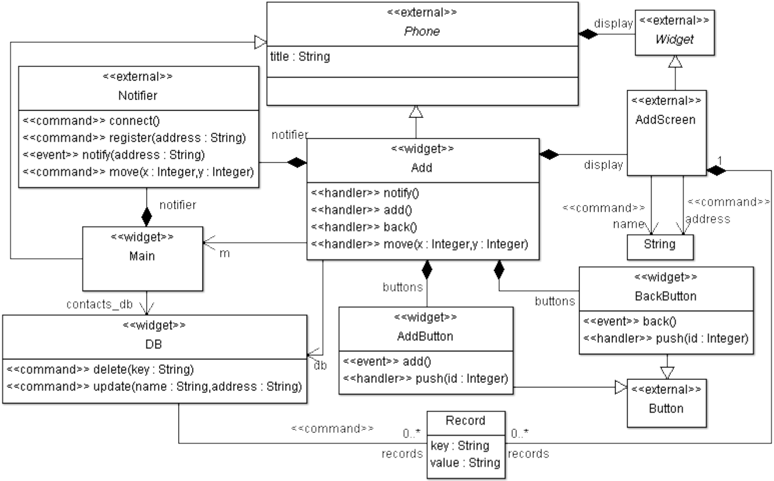
Contents

1. CARA Applications
   - Definitions
   - Research
   - Case Study
2. Modelling
   - Diagrams
   - Widget Calculus
3. Case Study Model
4. Other Issues
   - Types
   - Implementation

Main Screen


<<external>>
Window
title : String
<<event>> move(x : Integer,y : Integer)
<<external>>
Button
0..*
buttons
<<external>>
Phone
display
<<external>>
Widget
<<external>>
Clock
x : Integer
y : Integer
display
<<widget>>
Main
<<handler>> add()
<<handler>> del()
<<handler>> notify(address : String)
<<handler>> move(x : Integer,y : Integer)
<<external>>
Notifier
port : Integer
notifier
<<command>> connect()
<<command>> register(address : String)
<<event>> notify(address : String)
<<command>> move(x : Integer,y : Integer)
<<external>>
Widget
contacts_db
buttons
0..*
0..*
buttons
<<widget>>
DB
<<command>> delete(key : String)
<<command>> update(name : String,address : String)
<<widget>>
AddButton
<<event>> add()
<<handler>> push(id : Integer)
<<widget>>
DelButton
<<event>> del()
<<handler>> push(id : Integer)
<<command>>
0..*
records
<<external>>
Widget
Record
key : String
value : String
<<external>>
Button
label : String
<<event>> push(id : Integer)

Add New Contacts



<<external>>
Phone
title : String

display

<<external>>
Widget

<<external>>
Notifier
<<command>> connect()
<<command>> register(address : String)
<<event>> notify(address : String)
<<command>> move(x : Integer,y : Integer)

<<external>>
AddScreen

notifier

<<widget>>
Add
<<handler>> notify()
<<handler>> add()
<<handler>> back()
<<handler>> move(x : Integer,y : Integer)

display

1

notifier

<<command>>
name

<<command>>
address

<<widget>>
Main

m

String

buttons

buttons

<<widget>>
BackButton
<<event>> back()
<<handler>> push(id : Integer)

contacts_db

<<widget>>
DB
<<command>> delete(key : String)
<<command>> update(name : String,address : String)

db

<<widget>>
AddButton
<<event>> add()
<<handler>> push(id : Integer)

<<external>>
Button

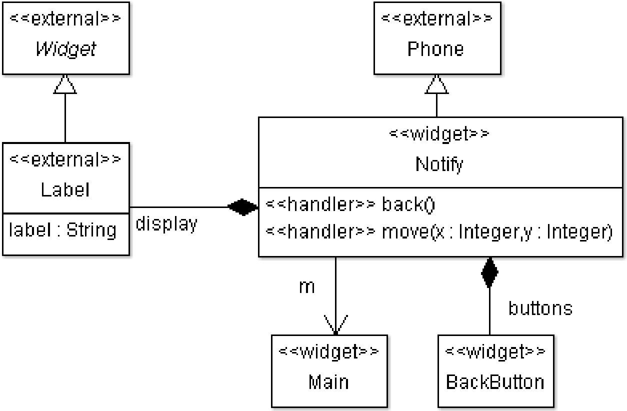
<<command>>

0..*
records

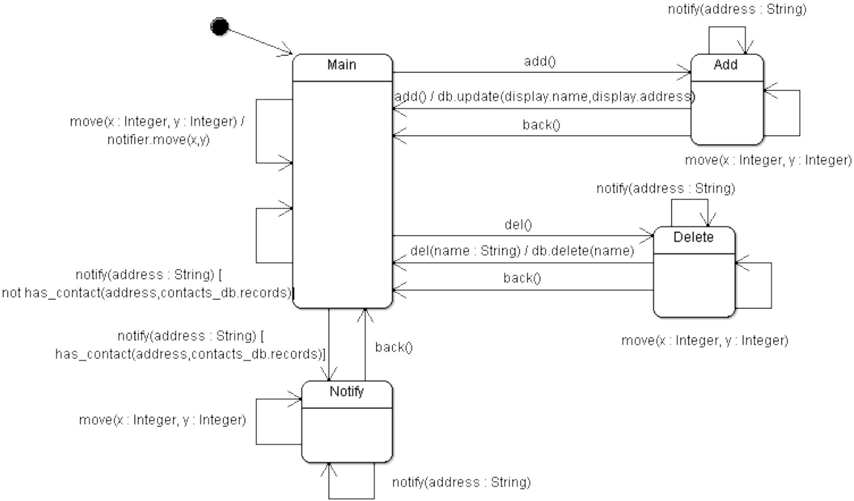
Record
key : String
value : String

0..*
records

Notify


<<external>>
Widget
<<external>>
Phone
<<external>>
Label
label : String
display
<<widget>>
Notify
<<handler>> back()
<<handler>> move(x : Integer,y : Integer)
m
buttons
<<widget>>
Main
<<widget>>
BackButton

Execution


notify(address : String)
Main
add()
Add
add() / db.update(display.name,display.address)
back()
move(x : Integer, y : Integer) /
notifier.move(x,y)
move(x : Integer, y : Integer)
notify(address : String)
del()
Delete
del(name : String) / db.delete(name)
back()
notify(address : String) [
not has_contact(address,contacts_db.records)]
notify(address : String) [
has_contact(address,contacts_db.records)]
back()
move(x : Integer, y : Integer)
Notify
move(x : Integer, y : Integer)
notify(address : String)

Contents

1. CARA Applications
   - Definitions
   - Research
   - Case Study
2. Modelling
   - Diagrams
   - Widget Calculus
3. Case Study Model
4. Other Issues
   - Types
   - Implementation

Widget Types

Standard functional types:

Atoms	`int`, `str`, `bool`
Functions	$\alpha \rightarrow \beta$
Lists	$[\alpha]$
Definitions	**type** `name =` α

Extra types:

Widget	`Widget`$(\alpha)\{x : \beta; \ldots\}$
Commands	$<\alpha>$
Events	$*$

Simple Button (with types)

```
external widget.Screen;
external widget.Button;

type Push = rec B.Widget(Button) { push:(int)-><B> }

type Main = rec M.Widget(Screen[<Push>]){ move:(int,int)-><M> }

rec val main:<Main> =
   fold[<Main>]
    widget self:Main (screen[<Push>](50,50,50,50,push)) {
     move(x:int,y:int):<Main> = do { return self }
    };
  val push:<Push> =
   fold[<Push>]
    widget self:Push (button(’PUSHME’)) {
     push(i:int):<Push> = do { return self }
    }
```

Java

- Java parser for language.
- Java type-checker for Widget.
- Java interpreter for Widget.
- Swing-based external library.
- Phone 'platform' in Swing.

Next Steps

- Implement the profile using standard UML tools.
- Link CARA models and Widget calculus via model transformations.
- More external libraries (Swing, HTML, Android).
- Simplify types.