Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Design-Aid for Graphical User Interfaces in Declarative Programming Languages

Ramon Gudschun

February 2011

Faculty of Engineering
Department of Computer Science
Programming Languages and Compiler Construction

Supervised by:
Prof. Dr. Michael Hanus

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den ________________________

Table of Contents

Illustration Index

1 Introduction

Modern applications often require a *graphical user interface* (*GUI*) to comfortably handle them. The developer of an application has to design and implement this interface. Most programming languages provide libraries for this purpose, but it is usually hard to imagine what the result of the corresponding source code will look like and if everything fits well together. Furthermore, the variety of options and graphical components, the so called *widgets*, such a library offers, complicates the development of *GUIs*. It can be aided by tools, the *GUI-builders*, which are available for some languages and *GUI*-libraries. It is possible to graphically design the interface and thus immediately get an impression of the final result, as well as an overview of the available *widgets*, with these tools. The output is generated source code, which corresponds to the design.

Applications can be divided into desktop applications and web applications. While desktop applications run on the local machine, web applications are distributed to multiple ones, where the access to clients is provided by a web browser. As the so called *look and feel*, defining the appearance and behavior of *widgets*, is related to that of the operating system for the former kind of applications and the browser's for the latter, *user interfaces* (*UIs*) should also be divided. In the following the term *GUI* is used for *UI*s of desktop applications and *web user interface* (*WUI*) for those of web applications.

Hanus introduced a *GUI*-library for the functional-logic programming language *Curry* in [Hanus 2000]. This approach has been extended and brought to a more abstract level by Hanus and Kluß in [Hanus, Kluß 2009], where the so called *UI*-library is described. Its advantage is that it is not related to a concrete type of *user interfaces*, but can be employed to define any type with a single definition.

The goal of this diploma thesis is to develop a *GUI-builder* for the *UI*-library in *Curry* (actually the term *UI-builder* is more appropriate, as *GUI*s and *WUI*s shall be supported). It shall realize the current standards for this kind of applications, although, in contrast to our implementation, just one kind of *UIs* is usually supported. Therefore, our process model prescribes an analysis of similar tools, where these standards can be derived and defined as requirements. This phase is followed by several extensions of the library, which are necessary to realize some of the requirements. The actual design and implementation covers the definition of an architecture and the concrete features related to the requirements and runs on an iterative and incremental basis. This procedure allows to implement features step by step and test each; hence, the system is consistent at any time. Due to the architecture, the generation of source code can be decoupled from the rest of the system. The final chapter deals with this subject.

2 Basics

In the following we provide short definitions for terms and abbreviations we employ in the later chapters. Furthermore, we introduce the basics of the programming language *Curry*, as well as some of its libraries. Finally, we examine the basic aspects of the scripting language *Tcl* and its framework *Tk* for *graphical user interfaces*.

2.1 Glossary

User Interface (UI): In computer science, a *user interface* is a system to control a program with input methods, like the keyboard or the mouse. The *UI* represents information graphical, textual or auditory [Wikipedia].

Graphical User Interface (GUI): A *graphical user interface* is a special type of *UI*, where information is graphically represented by *widgets*, in contrast to a pure textual representation. A *GUI* can usually be controlled by direct manipulation of *widgets* [Wikipedia].

Web User Interface (WUI): As a *web user interface* usually consists of graphical components, too, it actually is a subtype of a *GUI*, but is mainly used in web applications [Wikipedia]. We differentiate between a *GUI* for a desktop application and a *WUI* for a web application in this context.

Widget: A component in a *GUI*, like a button or a window, is called *widget*. A *widget* represents the program's data and provides an interaction point to directly manipulate it [Wikipedia].

Live Widget: There are two approaches to define a *widget* in a *GUI-builder*: either as a *live widget*, using the concrete *widget* from the *GUI*-library or as a simulation resp. a proxy of it, using other *widgets* [Molin et al. 1996]. The latter approach is sometimes taken if operations to configure the concrete *widget* and thus the *GUI*-design are not applicable. An example for such a proxy is a *widget* displaying just an image of the concrete *widget*.

Layout Manager: A *layout* or *geometry manager* is a system in a *GUI*-library to lay *widgets* out. In contrast to an absolute layout, where *widgets* are laid out pixel-wise, i.e., the user has to directly define positions and sizes, it provides a more abstract view, where positions and sizes are defined relative to each other [Wikipedia].

Look and Feel: The *look and feel* is a property of a *GUI*. Shapes, colors and layout define the look and the behavior of *widgets* defines the feel [Wikipedia]. If a *GUI*'s *look and feel* corresponds to the operating system's, it is called native.

Tcl/Tk: *Tcl* is a scripting language. The *Tk GUI* toolkit is an extension package for *Tcl*. As *Tcl* is frequently used together with this extension, it is often referred to as *Tcl/Tk* [Wikipedia].

Curry: *Curry* is a programming language, which combines two declarative paradigms: logic and functional programming. Its features are, for example, search, computing with partial information and efficient evaluation [Curry].

Integrated Development Environment (IDE): An *integrated development environment* is an application to aid software development. Part of an *IDE* usually is an editor for source code, a compiler and/or interpreter, as well as build tools and a debugger [Wikipedia].

UI-Builder and GUI-Builder: A *GUI-builder* is an application to graphically design a *GUI* by arranging *widgets*; hence, it is a *GUI* itself. The output is the *GUI*'s source code. An advantage over the direct definition of a *GUI* in code is that a user can immediately see the consequences of his design decisions, due to the *WYSIWYG* principle [Wikipedia]. In the following the term *UI-builder* is sometimes used to express that it may produce different kinds of *UIs*, not just *GUIs*.

What you see is what you get (WYSIWYG): *What you see is what you get* means that a system's content, which is graphically represented, corresponds to the final output [Wikipedia]. This principle is an important property of *GUI-builders*, where a user-defined design should look like the resulting *GUI*, which is generated by the builder.

Hypertext Markup Language (HTML): The *Hypertext Markup Language* is a markup language for web pages, which can be read by web browsers. An *HTML*-document's content is described by elements, the so called tags. A tag is enclosed in angle brackets (“<>”) and usually corresponds to a closing tag, which contains an additional leading slash. There can be text between an opening and a closing tag. A heading can, for example, be defined with the tag

```
<h1>This is a heading.</h1>
```

and the interpreter, usually a browser, displays the text with a big and bold font. A tag may also have attributes, to further define its properties, e.g.,

```
<h1 align=”center”>This is a centered heading.</h1>
```

to define a centered heading [Wikipedia].

Cascading Style Sheets (CSS): *Cascading Style Sheets* is a language to describe the presentation of a document defined in a markup language, especially *HTML*. Its main purpose is to separate a document's presentation from its content. Hence, while using the same document, the presentation may be changed by replacing just the style sheet. One or more styles can also be defined as a tag's attribute, resulting in a tag like

```
<h1 style=”text-align:center”>This is a centered heading.</h1>
```

for a centered heading again [Wikipedia].

JavaScript: *JavaScript* is an object-oriented scripting language. It is primarily used on the client-side of web applications in order to implement a *WUI* [Wikipedia].

Common Gateway Interface (CGI): The *Common Gateway Interface* allows to create interactive web pages by any programming language. A user's request to the web server is delegated to an application program, which returns a new web page in terms of an *HTML*-string [Wikipedia].

Extensible Markup Language (XML): The *Extensible Markup Language* is a markup language for documents. Like *HTML*, it employs tags and attributes to hierarchically describe a document. In contrast to *HTML*, it can be used for any kind of document, not just web pages, although a common use case is to describe a message, which is sent over the Internet. The tags and their attributes are defined by a developer [Wikipedia].

Command Pattern: The *command pattern* is a design pattern, where a command, i.e., a function or method, is encapsulated into an object. Implementing this pattern allows to set up a queue for commands, as well as logging and undoing them. The *client* creates the command object with the required parameters, especially the *receiver* to execute the command on. The object offers an interface to execute and undo itself and the *caller* does not have to know any details, except for this interface, to execute or undo a command. In order to realize undo, it is often necessary to save the *receiver*'s state in the command object before it is executed [Freeman et al. 2008].

Model View Controller (MVC): *Model View Controller* is a composite pattern. It consists of three components: the *model*, the *view* and the *controller*. The *model* contains the data and the application logic. It offers an interface to access and manipulate its state. The *view* represents the *model* to the user, e.g., in a *GUI*. The *controller* reacts on user interactions, i.e., handles events, with the *view* and accordingly changes the *model* and/or the *view*. Hence, the goal of *MVC* is to separate concerns from each other and thus it simplifies the implementation of and changes to each of its components.

A component may implement several patterns and the communication between components is often modeled with patterns, too. Therefore, the *model* is independent from *view* and *controller* and there may, for example, be different *views* for the same *model*. There can also be different *controllers* for one *view*, which can be replaced one another, e.g., one for an administrator and another for a normal user.

2.2 Curry Basics

The *UI-builder* we are developing in this work is implemented in the functional logic programming language *Curry*. In the following we would like to introduce the language's basic aspects, based on the [Curry tutorial], as well as the libraries for *user interfaces*. The most important features of declarative languages, namely nested expressions, lazy evaluation, higher-order functions, logic variables, partial data structures and search are covered by *Curry*. We cannot provide a complete definition of the language here; therefore, we focus on the features which are required below.

The language's implementation is distributed with a command line interpreter. There are different ones, but we stick to the most advanced, called *PAKCS* (Portland Aachen Kiel Curry System).

2.2.1 Modules

Although *PAKCS* provides an interactive environment, where simple commands can be directly evaluated, more complex programs have to be concluded in a *module*. A *module* is defined in a file with the extension ".curry". It can *import* other *modules*, in order to make their contents available to the current one. Furthermore, a *module* contains a set of *function* definitions. The predefined *module* `Prelude`, which defines the most frequently used *functions*, like `*`, `+`, `-`, `/` and `==`, `>`, `<`, `>=`, `<=`, `/=`, is always implicitly imported.

2.2.2 Functions

A *function* is defined by one or more *rules*, which have a *left-hand side* and a *right-hand side*, separated by an equals sign. The following example, with a single *rule*, computes the square of the given value:

```
square x = x * x
```

The *left-hand side*, `square x`, is a *pattern*, where a concrete call of the *function* `square`, e.g., "square 3", is tried to match to. If the argument of the call matches, its value is bound to the *pattern variable* `x`; thus, it can be employed on the *right-hand side*. After that, the expression on the *right-hand side*, `x * x`, is evaluated. A *function* f is usually evaluated by applying f on an expression e (e.g., a variable or another *function* call), i.e., "f e". In *Curry* a *function* can also be declared for infix application, in order to fit the usual notation. This is the case for the multiplication `*`, which is part of the `Prelude`.

A *function* can also be anonymously defined. The construct

```
\e1 -> e2
```

creates an *anonymous function* with arguments defined by `e1` and the *right-hand side* defined by `e2`. See below for an example.

The *right-hand side* of a *rule* can also contain a *conditional expression*. It has the form

```
if p then e1 else e2
```

where `p` denotes a predicate yielding either `True`, to evaluate `e1` or `False`, to evaluate `e2`. E.g.,

```
square x = if x == 1 then x else x * x
```

Another kind of conditional is the *case expression*. It has the basic form

```
case e of
  p1 -> e1
  p2 -> e2
  ...
```

where e is an expression, which may match to one of the *patterns* pi. If it matches, the corresponding expression ei is evaluated.

The *right-hand side* of a *function* may also contain a recursive call, i.e., a call of the *function* itself. The following example attempts to realize an iterative version of a *function*, which computes the power of the value x to n. The parameter p accumulates the current value and initially contains the radix x, too:

```
power x n p = if n > 1 then power x (n - 1) (p * x) else p
```

As *Curry* does not provide any control structures for loops, recursion is one of its core aspects.

The above definition of power is mathematically incorrect, as, for example, the call “power 3 0 3”, which should compute three to the power of zero, returns three and not one and the current definition does not take this case, where the exponent is zero, into account. We could just add another *conditional expression*, but let's try to solve this by using multiple *rules*, which cover the different cases:

```
power _ 0 _ = 1
power _ 1 p = p
power x n p = power x (n - 1) (p * x)
```

The *left-hand side* of a *rule* or a *pattern* of a *case expression* may not just contain variables, but also an expression for each argument, like the constants 0 and 1 in this case. An underscore denotes that any expression matches the *pattern* for this argument. It is called *anonymous variable*. Programming with multiple *rules* and concrete *patterns* is often easier than with, e.g., nested *conditional expressions*, because the desired cases can directly be written down. However, calling “power 3 0 3” again, might lead to a surprising result: There are different solutions. This is due to another feature, namely *non-determinism*. “power 3 0 3” matches to the *patterns* of the first and the third *rule*; hence, the interpreter can choose which one is evaluated. In general, by the order a *function*'s *rules* have been evaluated in, its results can differ, if more than one *pattern* matches the expression. If this behavior is undesirable, the *rules* have to be defined non-overlapping, i.e., with mutual exclusive expressions on their *left-hand sides*.

The *left-hand side* of a *rule* may contain a so called *guard*. This is a predicate, which further defines a *pattern*. If the *pattern* matches and the *guard* yields true, the *right-hand side* is evaluated. A *guard* is lead in by a pipe. We can add it to the example to distinguish the third *rule* from the others:

```
power _ 0 _         = 1
power _ 1 p         = p
power x n p | n > 1 = power x (n - 1) (p * x)
```

A *rule* may contain several *guards* with corresponding *right-hand sides*. We can add another one to the third *rule* to catch the case that the exponent is less than zero and print an error message (the *function* error from the Prelude stops execution and prints the given string):

```
power x n p | n > 1 = power x (n - 1) (p * x)
            | n < 0 = error "Invalid exponent!"
```

The definition of power is still a bit impractical, as the radix has to be passed in twice. We would like to hide this argument by using local definitions. There are two ways to locally define a function:

```
let e1 = e2 in e3
```

defines the expression `e2` as `e1` in the scope of `e3` and

```
where e1 = e2
```

defines `e2` as `e1` in the current scope. There can be multiple definitions in one *let clause* or *where clause* and clauses may be nested. Thus, the redefined example has the following form:

```
powerL x n =
    let power _ 0 _         = 1
        power _ 1 p         = p
        power x n p | n > 1 = power x (n - 1) (p * x)
                    | n < 0 = error "Invalid exponent!"
    in power x n x
```

or

```
powerW x n = power x n x
    where   power _ 0 _         = 1
            power _ 1 p         = p
            power x n p | n > 1 = power x (n - 1) (p * x)
                        | n < 0 = error "Invalid exponent!"
```

2.2.3 Types

So far, it may seem as if *Curry* was an untyped language. In fact, it is strongly typed, i.e., *functions* and arguments with the wrong type are detected by the compiler, but with the possibility to infer types. If a *function*'s definition does not provide a type declaration, the compiler's *type inference* algorithm computes a correct one, if possible. Nevertheless, it is better programming style to manually declare a type, at least for complex *functions*, as it contains information regarding the *function*'s usage.

A *data type* is a set of values. It is defined by

```
data t = v1 | v2 | …
```

where `t` is the name of the type and the `vi` are the different values it can take, i.e., the constructors, which are separated by a pipe. The `Prelude` predefines the most frequently used types, like `Bool` for booleans, `Int` for integers and `Char` for characters. If, e.g., `Bool` was not predefined, it could be as follows:

```
data Bool = True | False
```

Furthermore, a *data type* may be parameterized by adding one or more *type variables*. This is often useful in order to define an abstract data structure, which can hold different types of values, like a binary tree:

```
data BinTree a = Leaf | Node a (BinTree a) (BinTree a)
```

Hence, such a tree has a polymorphic type and can be parameterized with, e.g., `Int` or `Char` for `a`, to declare the type of its content.

A *function*'s type may now be declared by its name and a sequence of types for its arguments, as well as its return value, for example:

```
square :: Int -> Int
square x = x * x


setNodeValue :: BinTree a -> a -> BinTree a
setNodeValue Leaf _                       = Leaf
setNodeValue (Node _ left right) new      = Node new left right
```

The type declaration is started by `::` followed by the argument types and the type of the return value, separated by `->`.

Finally, frequently used *data types* can be abbreviated by declaring a *type synonym*, for example

```
type IntBinTree = BinTree Int
```

2.2.4 Lists and Tuples

A very important data structure in *Curry* is a *list*. *Lists* are built-in and can be created using a special syntax. A *list* either is *nil*, i.e., empty or *cons* and contains at least one element and a rest list, which may be *nil*. The constructor for *nil* is `[]` and for *cons* it is the infix operator `:`. A *list*'s type is the type of its elements enclosed in brackets, e.g., `[Char]`. The following *function* creates a *list* of ascending integers from zero to a given value:

```
nums :: Int -> Int -> [Int]
nums n max  | n >= max  = []
            | otherwise = n : nums (n + 1) max
```

The *function* `nums` recursively appends a *list* of numbers to the preceding one until the current one is greater or equal than the given maximum. Hence, the call "nums 1 5" creates the *list* `1 : (2 : (3 : (4 : [])))`. The constant *function* `otherwise` from the `Prelude` always yields `True` and is often used for the last one in a sequence of *guards*.

Functions on *lists* usually define one rule for *nil*, i.e., a *pattern* with the expression `[]` and another one for a non-empty *list*, i.e., an expression `(x : xs)`, where `x` is the *list*'s first element and `xs` the rest-list.

A *list* may alternatively be created by enclosing one or more elements, separated by commas, in brackets, e.g., `[1, 2, 3, 4]`. Furthermore, a *list* can be appended to another one by the operator `++`. Note that in *Curry* a string is just a *list* of characters; hence, `"Hello"` is equal to the *list* `['H', 'e', 'l', 'l', 'o']`.

A second important data structure is a *tuple*. In contrast to a *list* it has a constant length, but may contain values of different types. It is created by enclosing expressions in parenthesis, separated by commas, e.g.,

```
("Test", 123, 'a')
```

defines a triple of the type `(String, Int, Char)`.

2.2.5 Higher-Order Functions

In *Curry* a *function*'s argument may be a *function* itself, where arguments have not or just partially been applied to the latter. A *function* of the former kind is called *higher-order function*. This concept supports the implementation of flexible algorithms. The following (naive) sorting algorithm for a *list* takes an ordering criterion as an argument:

```
sort _ []       = []
sort f (x : xs) = insert f x (sort f xs)

insert _ x []                     = [x]
insert f x (y : ys) | f x y       = x : y : ys
                    | otherwise = y : insert f x ys
```

Hence, one can, for example, call "sort (<=) [3, 4, 2, 1]" for an ascending ordering or "sort (>=) [3, 4, 2, 1]" for a descending ordering, without redefining `sort`.

Higher-order functions are very common for the predefined *functions*, especially for the ones defined on *lists*, like `map` and `filter`:

```
inc :: [Int] -> [Int]
inc l = map (1 +) l

removeSpaces :: String -> String
removeSpaces s = filter (\c -> c /= ' ') s
```

`map` applies the given *function* on any element of the given *list* and `filter` retrieves a *list* from another one, with elements satisfying a given predicate.

2.2.6 Logic Variables

In contrast to a *pattern variable*, a *logic variable* occurs in the *guard* and/or the *right-hand side* of a *rule*, but not in the *left-hand side*. Therefore, it cannot be bound to a value by matching a *function* call, but its value is computed by the interpreter at runtime. A *logic variable* is locally declared by the keyword `free`, e.g.,

```
… where var free
```

It is evaluated either by *residuation* or *narrowing*. On the one hand, if v is a *logic variable* in the expression e, which cannot be evaluated, *residuation* suspends the evaluation of e until another expression has bound a value to v or the evaluation fails, if such an expression does not exist. For example, a call of the following *function* would suspend:

```
doubleR x | y == x + x = y
    where y free
```

On the other hand, *narrowing* guesses a value, which would allow evaluation to continue and binds it to v. The so called *constrained equality* =:= differs in two ways from the common boolean equality ==: Firstly, its return type is not `Bool`, but `Success`, with the related predefined operations `success` and `failed`, to denote whether guessing was successful or not. Secondly, it narrows instead of residuating. Hence, the following *function* would correctly double the given value:

```
doubleN x | y =:= x + x = y
    where y free
```

2.2.7 Input/Output

So far, we defined *functions* with a number of arguments, which yield the result of an application of expressions on these arguments. In contrast to imperative languages, such a *function* cannot have side effects, like changing the value of a global variable, i.e., it has no influence on its environment. But, if, e.g., input is read from the command line or the content of a file is changed, the environment, called *world* in this context, is changed, too. Therefore, if a *function* effects the *world*, it returns the new state. Such a *function* is called (I/O-) *action* and this concept *monadic I/O*. The changes take effect by implicitly applying a sequence of *actions* on the *world* and returning the type `IO t`, which is an abbreviation for

```
World -> (t, World)
```

and `t` is the actual result of the *function*, e.g., a file's content or `()`, if it is just the new *world*. Any *function* has implicit access to the *world*, without requiring an argument. For example, the predefined *action* `getChar`, which reads a single character from the standard input, has the type

```
getChar :: IO Char
```

and `putChar`, which prints a character to the standard output, has the type

```
putChar :: Char -> IO ()
```

Two *actions* can be combined to a sequence by the *function*

```
(>>) :: IO a -> IO b -> IO b
```

where the result of the first one is ignored, for example to print a string followed by a newline:

```
putStrLn []       = putChar '\n'
putStrLn (c : cs) = putChar c >> putStrLn cs
```

When the result of the first *action* is required in the second, the *function*

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

can be applied instead. Furthermore, the built-in *action*

```
return :: a -> IO a
```

just returns the argument, without changing the world and the abbreviation

```
done :: a -> IO ()
```

returns nothing.

More than two *actions* can be combined by repeatedly applying >> or >>=, but a more convenient way is the *do notation*. If `do` is applied on a number of vertically aligned *actions*, they are combined to a sequence, e.g.:

```
do putStrLn “Hello”
   putStrLn “World”
   putChar '!'
```

In a sequence of *actions* in the *do notation*, input can be stored in a variable by the operator <−, for example

```
do c <- getChar
   putChar c
```

which is just an abbreviation of

```
getChar >>= \c -> putChar c
```

The following example reads the file with the given name, appends the given string, writes the file back and returns its original content:

```
appendToFile :: String -> String -> IO String
appendToFile str file = do
    content <- readFile file
    writeFile file (content ++ str)
    return content
```

Sometimes it is necessary that *actions* share a variable holding a state, which is manipulated by a controlled side effect. This can be achieved with *IO-references*, defined in the *module* `IOExts`. They are used similar to files, but are not persistent and can hold any type of value and even a *function*.

```
newIORef :: a -> IO (IORef a)
```

creates a new *IO-reference* with an initial state,

```
readIORef :: IORef a -> IO a
```

retrieves the state of an *IO-reference* and

```
writeIORef :: IORef a -> a -> IO ()
```

updates the state.

2.2.8 GUI

In the following section we would like to introduce *Curry*'s library for *graphical user interfaces* [Hanus 2000], which is defined in the *module* `GUI`. In this library, a *GUI* consists of one or more *widgets*. These are organized in a hierarchical tree-like structure. The root of this hierarchy is a container or layout *widget*:

```
data Widget =
    Row [ConfCollection] [Widget]
  | Col [ConfCollection] [Widget]
  | Matrix [ConfCollection] [[Widget]]
  | ...
```

Such a *widget* can contain further *widgets*. The layout of these sub-*widgets* or children is based on a flexible grid: The available display space for the parent is distributed to cells and one cell is assigned to each child. A `Row` provides a single row of cells for a horizontal layout, a `Col` a column for a vertical layout and a `Matrix` a *list* of rows, which results in a table-like structure. Furthermore, a layout *widget* determines a *list* of configurations for any of its children, in order to align them, if a cell's size is greater than the *widget*'s:

```
data ConfCollection = CenterAlign | LeftAlign | TopAlign | ...
```

In contrast to a layout *widget*, a “primitive” *widget* cannot have children, but a list of `ConfItems`:

```
data Widget =
    PlainButton [ConfItem]
  | Entry [ConfItem]
  | Label [ConfItem]
  | ...
```

A `ConfItem` corresponds to a specific configuration option, e.g., simple ones, as a text label, width or reference, as well as an *event handler*, which is slightly more complex:

```
data ConfItem =
    Text String
  | Width Int
  | WRef WidgetRef
  | Handler Event (GuiPort -> IO [ReconfigureItem])
  | ...
```

References are based on *logic variables*, which are bound by *narrowing*; hence, in order to declare a *widget*'s reference, one just has to name it and declare it free.

An *event handler* is a *function* or *command*, which is called when an event, like pushing a button, occurs. The most common kinds of events are

```
data Event = DefaultEvent | MouseButton1 | MouseButton3 | ...
```

where a `DefaultEvent` is the standard event of a *widget*, as the button-push mentioned above. The handler's argument, the `GuiPort`, is automatically passed to the handler when it is called by the scheduler. It provides an interface to the communication stream with *Tcl/Tk*, on which the library is based. Because an *event handler* impacts the *world*, its return type is `IO`, together with a *list* of `ReconfigureItems`. An *event handler* may write something to a file or a database, but a common use case also is to apply changes on *widgets*. This is what the `ReconfigureItems` are used for. There are different constructors for this type, but usually just the following is required:

```
data ReconfigureItem = WidgetConf WidgetRef ConfItem
```

Thus, a reconfiguration of a *widget* is performed by creating one or more `ReconfigureItems` with the *widget*'s reference and the new property and returning these in an *event handler*.

Furthermore, the *function*

```
getValue :: WidgetRef -> GuiPort -> IO String
```

retrieves a *widget*'s current value, which is related to the kind of the *widget* and

```
setValue :: WidgetRef -> String -> GuiPort -> IO ()
```

sets it. For a `PlainButton` this is its text label and for an `Entry` the current text.

Finally,

```
runGUI :: String -> Widget -> IO ()
```

runs the given *widget* in a new window with the given string as its title.

The following example shows a simple counter *GUI*:

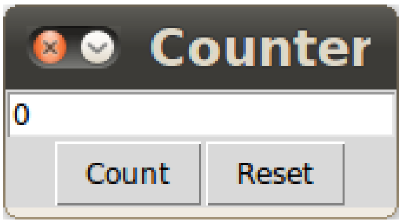

Illustration 1: A simple counter GUI

```
import GUI
import Read

widget = Col [] [
        Entry [WRef counter, Text "0"],
        Row [] [
            PlainButton [Text "Count", Handler DefaultEvent count],
            PlainButton [Text "Reset", Handler DefaultEvent reset]
        ]
    ]
    where   counter free
            count gp    = do
                val <- getValue counter gp
                setValue counter (show ((readInt val) + 1)) gp
                return []
            reset _     = return [WidgetConf counter (Text "0")]

main = runGUI "Counter GUI" widget
```

The `Entry` with the reference `counter` is controlled by two `PlainButtons`. For this purpose each one has an *event handler* assigned to it. In the first *command*, the current text of the `Entry` is retrieved by `getValue`. The text is casted to an integer by the *function* `readInt` from the *module* `Read`, incremented, casted back to a string by `show` and then assigned to the `Entry` by calling `setValue`. As there are no `ReconfigureItems`, an empty *list* is returned. In the second *command* a `ReconfigureItem` with the new text is created and returned, instead of calling `setValue`.

2.2.9 UI

Besides the *GUI*-library, a more abstract approach for *UIs* has been proposed by [Hanus, Kluß 2009]. Its main purpose is to provide a general definition to describe any kind of *UI*. The structure, defined by the *module* `UI`, is similar to that of the *module* `GUI`. There are different kinds of *widgets*, like buttons and text fields, organized in a grid-based layout and corresponding layout *widgets*, but instead of a concrete constructor for each *widget*, a generic one is used:

```
data Widget r act1 act2 =
  Widget
    (WidgetKind r act1 act2)
    (Maybe String) (Maybe (Ref r)) [Handler act1 act2]
    [StyleClass]
    [Widget r act1 act2]
```

Hence, any *widget* basically has the same structure. The constructor's parameters are:

1. The kind of the *widget*, for example `Button` or `Row`.
2. A text label (the type `Maybe` from the `Prelude` either is created by `Just v`, if the value `v` shall be defined or `Nothing`).
3. A reference. There is no special constructor required to create one, except for `Just`.
4. A *list* of *event handlers*. Defining a handler is slightly different than in the *GUI*-library, as the *command* is created by the additional constructor `Cmd`.
5. A *list* of style classes. A `StyleClass` is a *list* of `Styles` and a `Style` configures a *widget* regarding its appearance and layout, e.g., `Fill X` to horizontally stretch it to the size of its cell or `Bg Blue` to set its background to blue.
6. A *list* of children.

The types of the *type variables* `r`, `act1` and `act2` are determined by the *modules* providing the concrete implementation (see below), e.g., the type of a reference or the type of an *event handler*.

For the most common *widgets* and use cases abbreviations of this constructor exist, for example

```
col :: [Widget r a1 a2] -> Widget r a1 a2
```

to create a `Col` from a *list* of children or

```
label :: String -> Widget r a1 a2
```

to create a `Label` with the given text. However, as, for example, references and handlers are not taken into account by every *function*, we will usually stick to the generic constructor in the context of this work.

Furthermore, *functions* to run the *UI* and dynamically change a *widget*'s configuration have been abstractly implemented, like `runUI` (replaces `runGUI`), `getValue` (as before), `setHandler` (sets a new *event handler*) and `changeStyles` (applies style classes).

In order to define and finally run a *UI*, a concrete implementation of these abstract definitions has to be provided. So far, there are implementations for a *WUI* in the *module* `UI2HTML`, which is based on *HTML* and *JavaScript* and a *GUI* in the *module* `UI2GUI`, which is again based on the *module* `GUI`. Illustration 2 shows the structure.

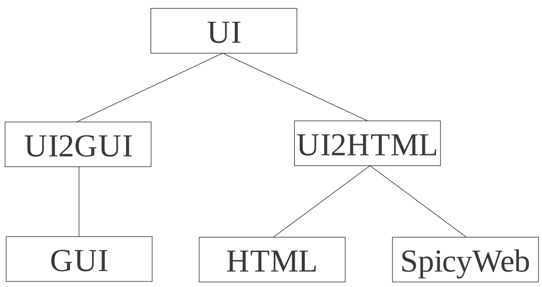

Illustration 2: The structure of the UI-library

The advantage of the abstract approach is revealed when a concrete *UI* is implemented. The following example redefines the counter *GUI* from above as a *UI*:

```
import UI2HTML --UI2GUI
import Read

widget = col [
        entry counter "0",
        row [
            button count "Count",
            button reset "Reset"
        ]
    ]
    where   counter free
            count env   = do
                val <- getValue counter env
                setValue counter (show ((readInt val) + 1)) env
            reset env   = setValue counter "0" env

main = runUI "Counter UI" widget
```

Illustration 3: The counter UI as a WUI

The abbreviations have been employed instead of the generic constructor here. Using the latter, the first button, for example, would be implemented as follows:

```
Widget Button (Just "Count") Nothing [Handler DefaultEvent (Cmd count)]
  [] []
```

In order to define the *UI* either as a *GUI* or a *WUI*, the corresponding *module* has to be imported, like `UI2HTML` for a *WUI*, in this case. The rest of the source code stays untouched. To finish a *WUI* definition, the command line tool `makecurrycgi` then creates a *CGI*-script from the *.curry-file, which can be run by a web server and accessed by a web browser.

Illustration 3 shows the result as a *WUI*. The *GUI*-version exactly looks as in Illustration 1.

2.3 Tcl/Tk Basics

Because the module `GUI` is based on *Tcl/Tk*, we would like to give a short introduction to the latter, based on the [Tk tutorial]. There are three important concepts in *Tk*: *widgets*, *layout management* and event handling.

2.3.1 Widgets

Tk offers classes for many different kinds of *widgets*, like buttons, labels, entries and frames. Most of them have also been implemented in `GUI` and `UI`. In *Tk* they are also organized in a hierarchy. The root window is always at the top and may contain several *widgets* and content frames. Each content frame may again contain a number of *widgets*, as well as further content frames.

In order to create a *widget*, one has to declare its class and label. The label also represents the hierarchical structure, where a parent-child relationship is denoted by a dot. The root window always has the label ".". Hence, for example a button as a child of the root window can be created by

```
button .b
```

If we would add it to a frame before, this would be declared as

```
frame .f
button .f.b
```

The outcome of such a declaration is a so called *object command*, where methods to communicate with the *widget* can be called on.

Related to its class, a *widget* can be configured by a sequence of *configuration options*, e.g., a button's text, width or color. This can either directly be done when the *widget* is created or by calling `configure` on the *object command*:

```
button .b –text “Ping”
.b configure –text “Pong”
```

2.3.2 Layout

By declaring and configuring a *widget*, it will not yet be displayed. To achieve this, its size and position on the screen has to be determined by a *layout manager*. Each content frame has its own; thus, layouts can be nested. There are several kinds of *layout managers* in *Tk*, but we stick to one called `grid`. This is a method, which can be called on *object commands*, together with options for the specific layout, e.g.,

```
grid .b –row 1 –column 1
```

The *layout manager* arranges the available space in a grid, where each *widget* is placed in one or more cells. It is common to use “grid” as a verb, i.e., saying “to grid”. Important options for `grid` are:

- `row` determines the row position in the grid.
- `column` determines the column position in the grid.
- `rowspan` determines the number of rows a *widget* spans in the grid.
- `columnspan` determines the number of columns a *widget* spans in the grid.
- If a *widget* is smaller than its cell(s), `sticky` defines what to do with the remaining space. A combination of directions, i.e., w(est), n(orth), e(ast) and s(outh), is assigned to the option to define an orientation. Two opposite directions can be combined, in order to make the *widget* fill the available space horizontally or vertically. A combination of all directions makes the *widget* fill all the available space.

2.3.3 Event Handling

In order to define how a *widget* reacts on certain events, an *event handler* or *callback* has to be assigned. For the usual default event, which can occur on a widget, e.g., pushing a button, this is possible via the *command option*. For example

```
button .b –text “Ping” –command {.b configure –text “Pong”}
```

changes the button's text from “Ping” to “Pong”, when it is pushed (we anonymously define the *callback* in braces (“{}”) here, but it is, of course, also possible to define a procedure and assign its name, but we do not go into details regarding procedures here). For other events than the default, the `bind` command is used:

```
bind .b <3> {.b configure –text “Ping”}
```

This example binds the event `<3>`, which corresponds to a right mouse button click, to an *event handler*, which sets the button's text to “Ping” again.

3 Analyzing GUI-Builders

In this chapter we analyze and evaluate existing *GUI-builders*. With the results we will be able to emerge requirements for an own builder. These are common features, we think a *GUI-builder* must have and a user expects to find in such an application, as well as some special nice-to-have features.

We have chosen two popular *IDE*s with *GUI-builders* and one standalone application for the analysis. These are:

- The NetBeans IDE [NetBeans], which has especially been designed for Java and its *GUI*-framework Swing, but also supports languages like C and C++.
- Visual Studio 2010 Express [Visual Studio] for C++. The regular versions support multiple languages, the express editions a single one each. We have chosen the C++-version.
- GUI Builder [GUI Builder] is a standalone builder for *Tcl* and the *Tk*-framework. There is support for other languages as well. It is related to the SpecTcl project.

The latter fits our purposes well, as *Curry*'s *GUI*-library and GUI Builder both are based on *Tcl/Tk* and thus have similar requirements regarding layout, *widget* types and their properties.

We had also planned to examine SharpDevelop [#develop], an *IDE* that has been designed for C# and Basic. But it turned out to be too similar to Visual Studio and therefore would not provide any further knowledge.

We employ a set of criteria for the evaluation, see next section. With each *GUI-builder* we also design a simple *GUI*, in order to check how well standard use cases can be realized. As an example, we create a calculator, which should look similar to the following sketch:

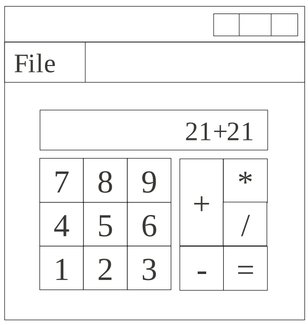

Illustration 4: An example GUI

3.1 Criteria

We are now defining a simple set of criteria or categories for the different aspects of a *GUI-builder*, inspired by [Molin et al. 1996]. Based on these, we evaluate the builders introduced before.

3.1.1 Layout

We examine which kinds of layouts the builder supports and give a rough description. As this strongly depends on the underlying *GUI*-library, we put more weight on the usability of these layouts, like the possibility to nest them and access *widgets* in these, as well as their general configuration.

3.1.2 Core Features

These are basic features, like the placement of *widgets* and accessing and changing their properties.

The goal is to examine features which are absolutely necessary to design a *GUI*. Also, as menus are likely to be different from common *widgets*, we are especially interested in how the builders handle these. Another important aspect is how event handling can be managed.

3.1.3 Advanced Features

We examine whether there are uncommon, but useful features, as well as features not really required to design a *GUI*, but nice to have or probably expected by users. These cover direct manipulation of *widgets*, drag and drop and the design and import of custom *widgets*.

3.1.4 Conclusion

In the conclusion we sum up the overall structure and usability of the application. A typical question could be, if there is a chance a user gets lost in the amount of possibilities or if the program guides him through the steps of designing the *GUI*. We also want to know if the resulting *GUI* is identical to the one designed, i.e., if the *WYSIWYG* principle applies. We furthermore describe what we like and what we dislike about the application, in terms of usability and usefulness. This is however quite subjective.

3.2 NetBeans

When starting NetBeans the first time, we can, after creating a new Java project, choose whether we want to design a *JFrame* (a main window) or a *JPanel* (a standard container). Both options lead to the following view:

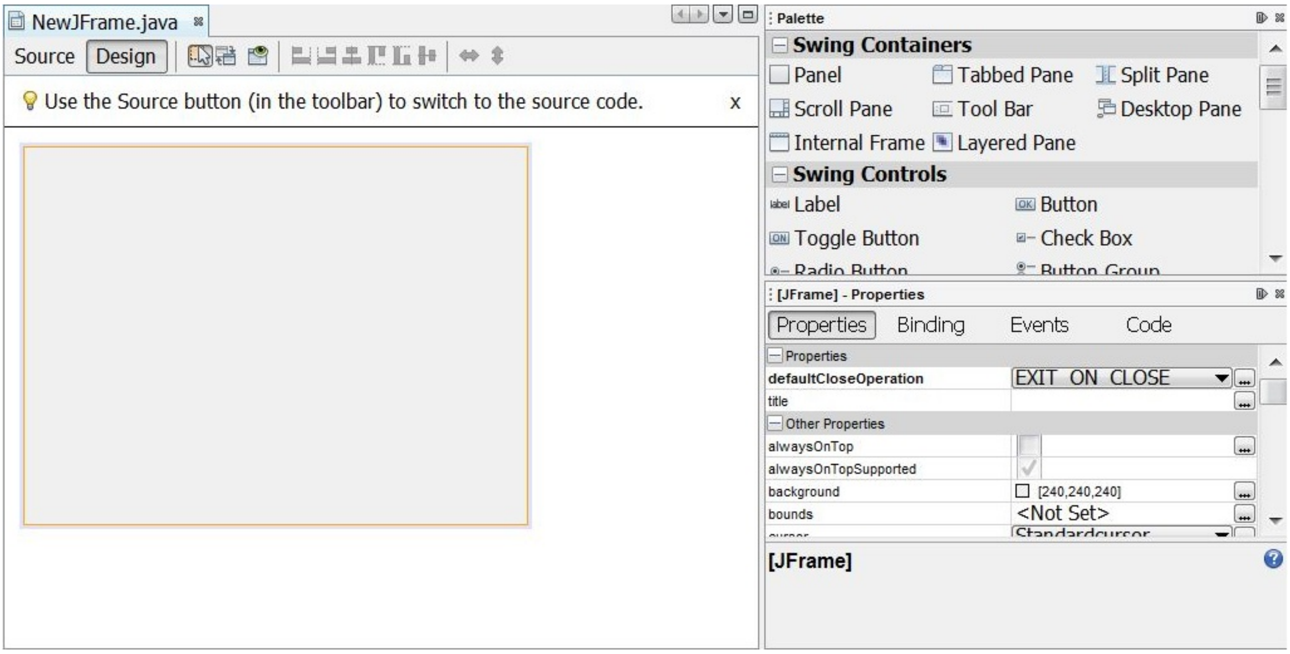

Illustration 5: Designing a main window in NetBeans (we cut out irrelevant parts of the IDE)

On the left is the *Workspace* with the tool bar, where *widgets* can be placed and arranged. On the top right is a list of *widgets* with symbolic icons, the *Palette* and below the *Properties* view for the current *widget*. The *Properties* view contains four tabs; we focus on the *Properties* and the *Events* tab here. Below these tabs, at the bottom of our screenshot, is a field which is providing some additional information about the current selection, like a *widget*'s or property's name and type and in case of the latter, a short description.

There is also a view with a tree-like structure, called *Inspector*, not covered by the screenshot, showing the *widget* hierarchy and the corresponding layouts.

One can switch between the design view and the generated source code in the tool bar. One can also choose between the *Selection Mode* (the default) and the *Connection Mode* (see 3.2.3). Additionally, there are some buttons in the tool bar to change a *widget*'s layout (see 3.2.1).

The other components are described below.

3.2.1 Layout

There are several types of *layout managers* available in NetBeans. We present the two more powerful ones.

The default layout in the *GUI-builder* is the *Free Design*. When we examine the generated code, we find out that this actually is the *GroupLayout*. *Widgets* are aligned and sized relative to each other. If *widgets* are of equal size, these changes can be solely applied by direct manipulation (see 3.2.3). The alignment of different sized *widgets* can be adjusted with the tool bar: Let's say there is a tall and a flat *widget* side by side. When both of them are selected, the buttons in the tool bar become active. By pushing one of the corresponding buttons, the flat *widget* can then be aligned towards the top, bottom or center of the taller.

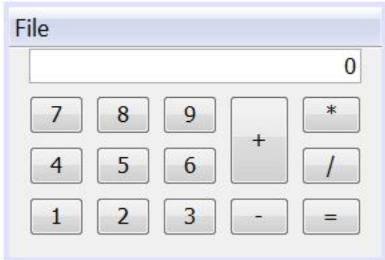

Illustration 6: The example GUI designed with the Free Design

The *GroupLayout* allows a rapid design, but lacks options regarding customization. We have not been able to remove the spacing the *layout manager* adds between *widgets* (see Illustration 6). Also, when changing a single *widget* in size or position, there are sometimes weird changes to other *widgets*, so the layout for these has to be repeated.

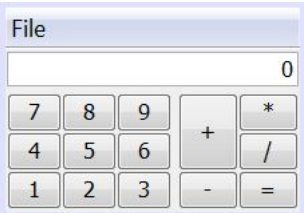

Illustration 7: The example GUI designed with the GridBagLayout

The second layout type we examine is the *GridBagLayout* (see Illustration 7). *Widgets* are laid out based on a flexible grid, similar to *Tcl/Tk* and *Curry*. A *widget* can just take a part of its grid cell or also several cells. The widest *widget* defines the width of cells in that column and the highest the height of cells in that row. Parameters, the so called *GridBagConstraints*, like the position in the grid and the grid width for a *widget*, can be applied via the *widget*'s properties.

A disadvantage when laying out with the *GridBagLayout* is that there is no way to directly manipulate *widgets* in the *Workspace*. To choose, for example, a *widget*'s cell, its properties have to be edited. Hence, one has to know what a single *GridBagConstraint* exactly does. These disadvantages can however be neglected for the most part when using the *GridBagLayout Customizer* (see 3.2.3).

Layouts can also be nested into each other.

3.2.2 Core Features

One way to add a *widget* to another one is via the container's context menu, which is accessible by a right mouse button click. The category and the *widget* are chosen in a sub-menu. Depending on the *layout manager*, the *widget* will then be placed somewhere in the *Workspace*. A single *widget* can then be selected with the mouse, a colored frame is put on it and its properties show up in the corresponding tab in the *Properties* view. The properties can also be accessed by the context menu in a separate dialog.

The properties are simple key-value pairs, but tailored to the concrete *widget*, without properties which cannot be applied. The values are strings. They can be edited in text fields and some also provide a drop-down menu with possibilities to choose from.

Menus (*MenuBar*, *Menu* and *MenuItem*) in NetBeans are more or less treated like other *widgets* (see 3.2.3).

Similar to properties, events can be defined via *Properties → Events*. These are key-value pairs again, where the key is an event which can occur on this *widget* and the value is the name of the event handling method.

3.2.3 Advanced Features

NetBeans provides a preview function for the current design. In our tests it had the same look as the design and we could click on buttons or edit text fields, but further event handling was disabled.

Another way to add a *widget* to the design is by dragging it from the *Palette* and drop it in the *Workspace*. For a better accessibility, the *Palette* is divided into categories like *Swing Containers* and *Swing Controls*. These categories can be faded out, together with their entries.

It is possible to select multiple *widgets* at the same time. Properties they share are displayed in the *Properties* view. When they differ in a value, this is denoted by “<Different Values>”. Anyway, these can be changed for the selected *widgets* together, i.e., we can set the property *text* for a button and a text field for example. Also, like files in an operating system or text in a word processor, selected *widgets* can be copied, cut and pasted.

When using the *Free Design*, *widgets* can be changed in size and position by direct manipulation. Selecting a *widget* then adds eight controls for the size to the highlighting frame. Direct manipulation in the *Workspace* is not possible when using the *GridBagLayout*, but in the *GridBagLayout Customizer*. This is a separate dialog showing the grid with the *widgets*, the selected *widget*'s layout properties, as well as controls for most of the *GridBagConstraints*, see Illustration 8.

In order to add a new column or row to the layout, either the *Grid X* and *Grid Y* properties can be set or a *widget* dragged to an empty position. Due to the *Customizer*, the quite complex *GridBagLayout* becomes well usable – we could easily define the layout for our example *GUI*.

Customizing *widgets* is also possible via the hierarchy. This tree-like structure, called *Inspector*, contains the *widgets* used in the design, together with their names and types, as well as *layout managers* and provides another way to access the properties. It is especially useful when layouts have been nested.

Every change the user made in the *Workspace*, the *Properties*, the *Inspector* and even the *GridBagLayout Customizer* can be undone.

A powerful feature is the *Connection Mode*, which is accessible via the tool bar. Instead of selecting *widgets* in order to change their properties, a source and a target *widget* are selected. With the help of a wizard, an event on the source is selected and a name for an event handling method defined. In the next step what to do with the target when this event occurs is set up. Either a property is changed, a method called or custom code provided. Finally, the parameters are defined in case of the former two options.

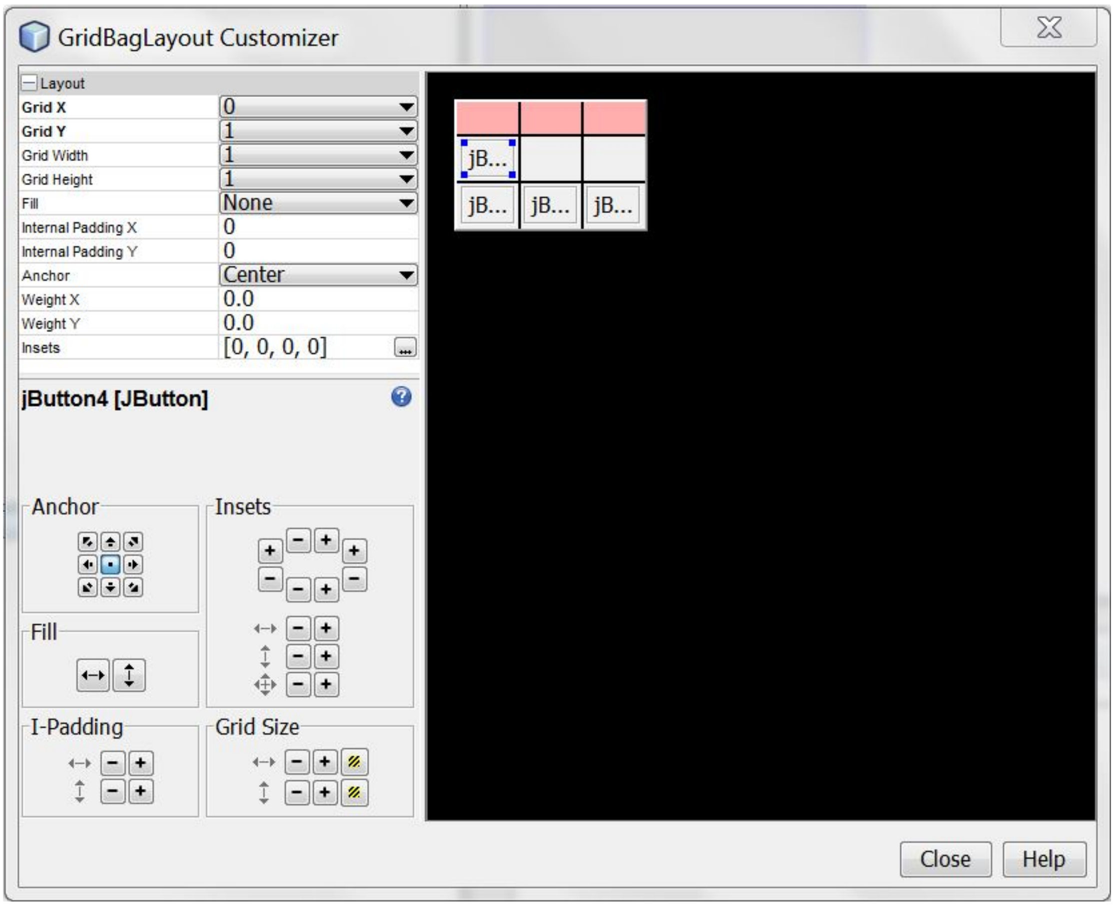

Illustration 8: The GridBagLayout Customizer

Menus are treated like other *widgets* in NetBeans. But there is some additional functionality: A menu item's icon and short cut can be directly changed when the menu in the *Workspace* is opened (see Illustration 9). A file open dialog is used to choose an image file for the icon and left clicking the short cut label provides a wizard to configure this short cut.

Finally, there is an easy way to employ custom *widgets* in the *GUI-builder*. User defined *widgets* can be added to the palette and used like any of the standard *widgets*. We can, for example, select the *widgets* forming our number pad in the example *GUI*-design, put them into a *JPanel* (by copy and paste) and add this *JPanel* to the *Palette*. After that, we can insert the pad as a single *widget* into the *Workspace*.

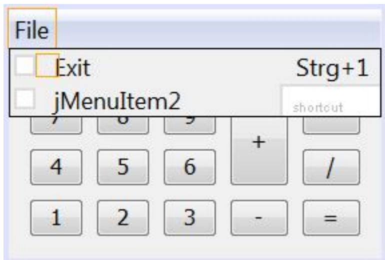

Illustration 9: Designing a menu

3.2.4 Conclusion

We have examined two *layout managers* in NetBeans. On the one hand, the *Free Design* allows for a rapid design and uses direct manipulation to arrange *widgets* in an intuitive manner. But the lack of customization options fails when it comes to design details. Also, because *widgets* are related to each other, a change to one *widget* often leads to unexpected changes to other *widgets* and thus complicates the design. The *GridBagLayout*, on the other hand, is less intuitive at the first glance, as it does not allow direct manipulation. However, when using the *GridBagLayout Customizer*, the layout can easily be configured. We think that the *GridBagLayout* is the overall better and more powerful solution. It would probably advance further, when the *Customizer* would be directly integrated into the *Workspace* or the main view.

There are different ways to access a *widget*'s properties and they are where we would expect them to be located. The *Palette* and the *Properties* can be a bit confusing, but fading out categories helps. This is also due to the many available options.

Assigning event handlers has been solved like properties and as we would expect it. A very useful feature seems to be the *Connection Mode*. In this mode we are able to define a complete event handler, without directly accessing the code.

The possibility to select multiple *widgets* at once, together with the copy, cut and paste features, is a useful addition. This assists the user when a bigger amount of similar *widgets* has to be laid out.

Layouts can be nested and *widgets* deeper in the hierarchy, which may be difficult to select, can be accessed via the *Inspector*. Another way to achieve a more complex design is with custom *widgets*. These are very easy to handle in NetBeans.

3.3 Visual Studio

In Visual Studio we start with the choice of the project type. In order to design a *GUI*, we choose a *Windows Forms Application*. An initial design for a main window opens:

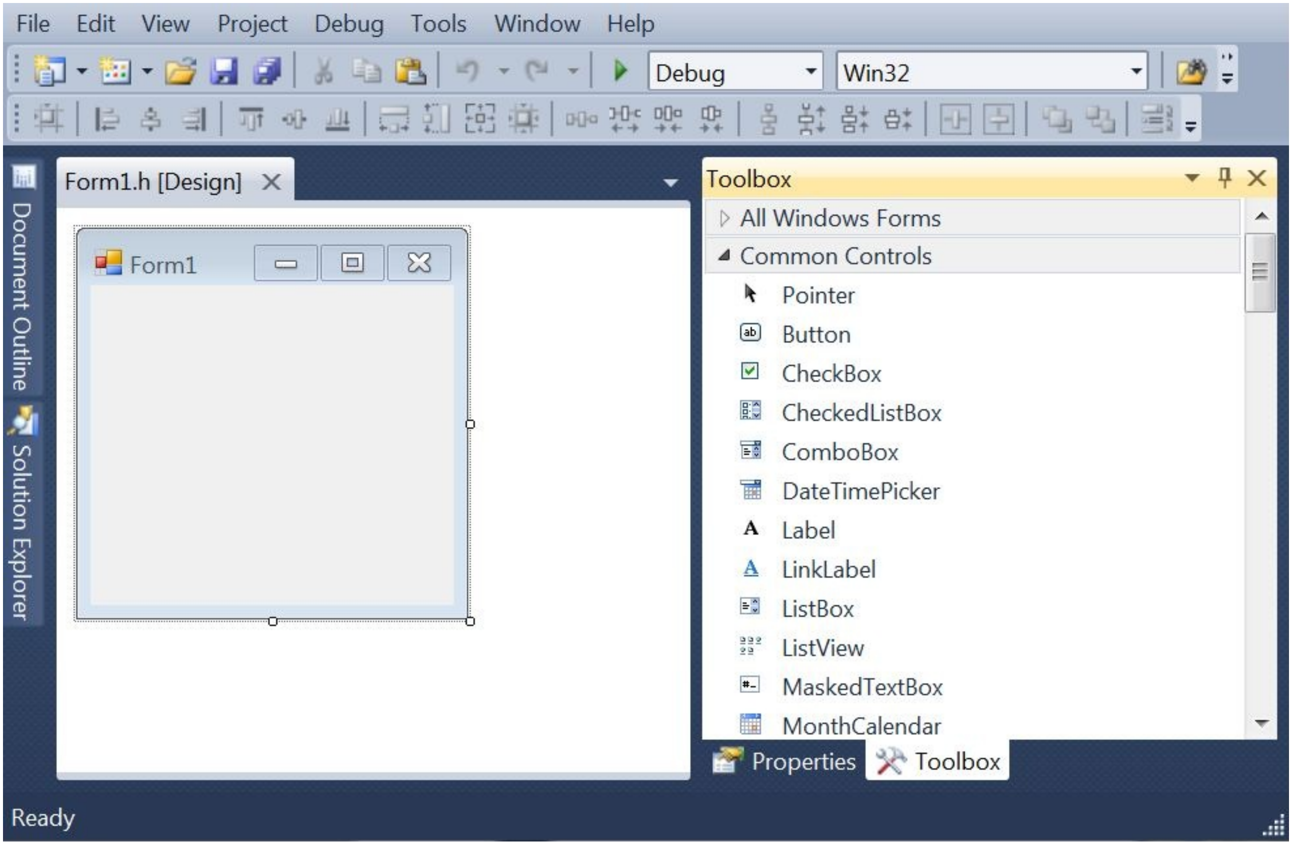

Illustration 10: Designing a main window in Visual Studio

Located at the top of the screenshot are the main menu and the general tool bar of the *IDE*, as well as the *GUI-builder*'s tool bar. Below these and on the left is the *Document Outline* (faded out) and the design view and on the right the *Toolbox* and the *Properties*. One can switch between different tabs in the *Properties* view. We focus on the *Properties* and the *Events* tabs here.

The tool bar contains controls to directly manipulate the design's layout, like the alignment of or the margins (insets) between *widgets*. We examine most of them in 3.3.1.

The *Document Outline* is a hierarchy view with a list of *widgets* which have been used in the design and the *Toolbox* provides a list of *widgets* which can be added to the design, both similar to the corresponding views in NetBeans.

3.3.1 Layout

Initially, there are three different *layout managers* in Visual Studio. The standard layout is nearly identical to the *GroupLayout* in NetBeans. It is always applied to the top-level window which is designed and cannot be replaced. In order to add one of the other *layout managers*, a container *widget* has to be added, either the *FlowLayoutPanel* or the *TableLayoutPanel*. There are some more containers, like the *SplitContainer*, providing a kind of layout, but we don't really consider these „layout managers", due to their simplicity. It is also possible to add additional *widgets* from the library, including containers, to the *Toolbox*.

The standard layout is relative, like the *GroupLayout*. It is however configurable with the tool bar, if two or more *widgets* have been selected. The horizontal spacing between *widgets* can be equalized, increased, decreased or removed there. This accordingly applies to vertical spacing. Different sized *widgets* can also be aligned, like in NetBeans.

The *FlowLayoutPanel* provides a very simple layout, where *widgets* are appended to a row. If there is not enough space available in one row, the following *widgets* are laid out in the next row below, starting on the left and so on. Margins can be adjusted in the *widgets*' properties. The *FlowLayoutPanel* is however too simple for the employment in an advanced design.

The *TableLayoutPanel* is similar to the *GridBagLayout*, which has an underlying grid structure, but is a *widget* itself. The size of the panel can be manually adjusted or its *AutoSize* and *AutoSizeMode* properties can be set. In case of the latter the panel will adjust its size according to the *widgets* it contains. As any other *widget*, it has to be aligned in the parent window. Either the *Center Horizontally* and *Center Vertically* buttons in the tool bar are used or it is dragged to the desired position. Rows and columns may be added or removed to or from the *TableLayoutPanel* in a context menu. For further customization the *Column and Row Styles Dialog* has to be used, which is also available in the context menu (see Illustration 11). Rows and columns are separately configured there. There are three options regarding the row's or column's size type:

1. *Absolute*: Adjusts a value for the desired number of pixels a row shall be high or a column shall be wide.
2. *Percent*: Assigns the fraction of the available space a row or column shall take.
3. *AutoSize:* The size of the row or column is automatically adjusted.

We use the *Absolute* size type for our example *GUI*. It would be ideal to use the *AutoSize* option and thus let the layout manager automatically lay out as far as possible, but there is one problem: The option equally divides available space between rows and columns. Hence, even with equal size properties, the plus and minus buttons become smaller than the other widgets, due to the margin we assign to them. That's why we apply a slightly greater absolute value to the fourth column, as also shown in Illustration 11.

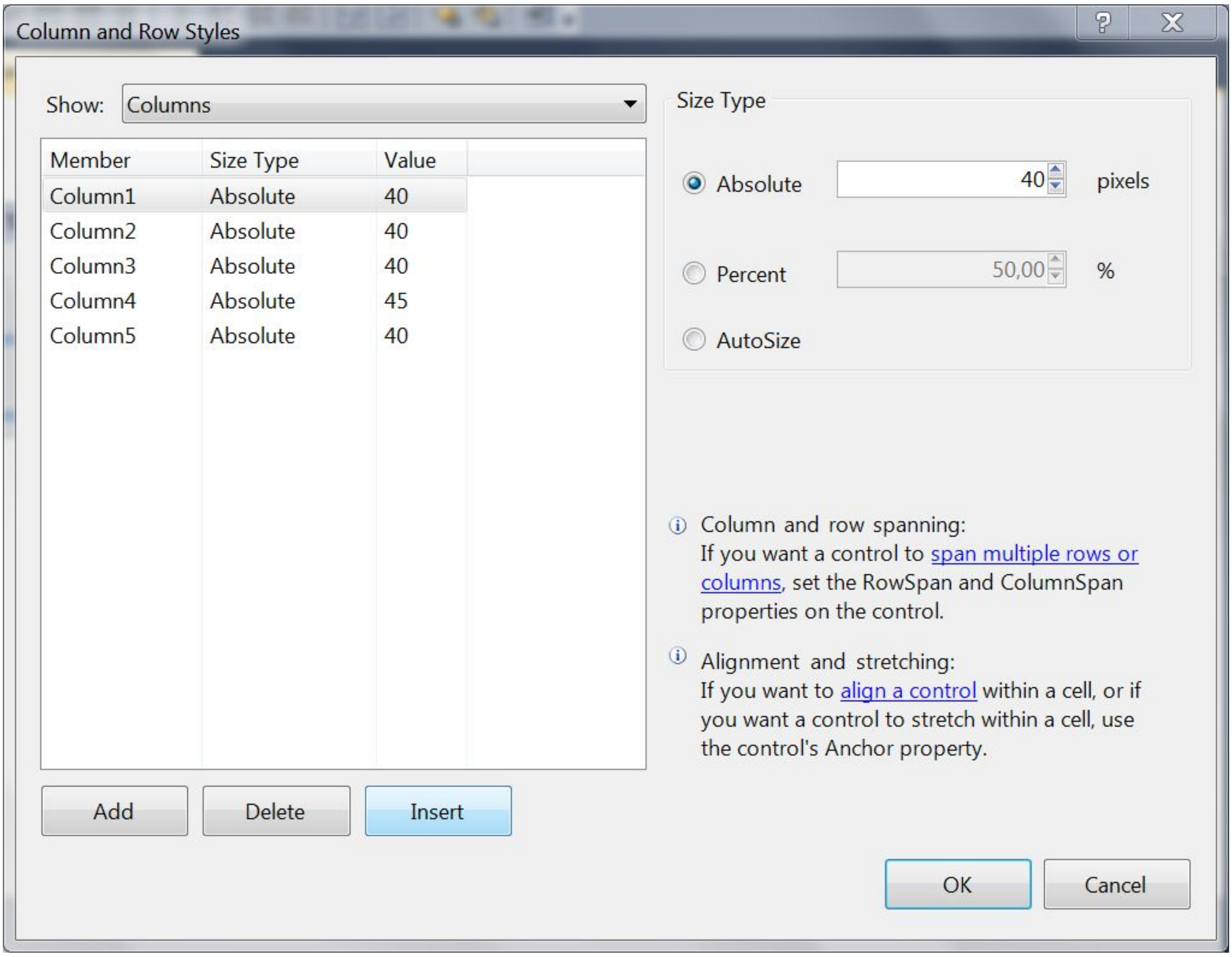

Illustration 11: Customizing the TableLayoutPanel

The final result for the example is shown in Illustration 12. The design with the default layout is identical, but without the empty space at the bottom. If we would drag the window flatter, the *TableLayoutPanel* would move above the menu bar, covering it. We did not experience this problem with the default layout.

Another disadvantage with the *TableLayoutPanel* is that important layout parameters, like *Column* and *Row Span*, *Anchor*, *Dock* (fill) and *Margin*, cannot be changed in the *Column and Row Styles Dialog*, so we have to switch between this and the *Properties*.

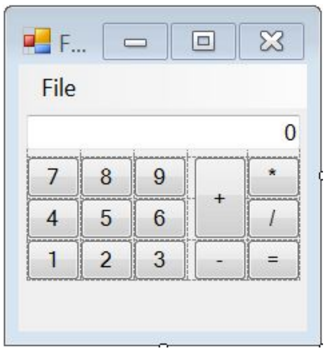

Illustration 12: The example GUI designed with the TableLayoutPanel

3.3.2 Core Features

In Visual Studio *widgets*, including menus, can solely be added to the design by drag and drop from the *Toolbox* to the desired location. Thus, drag and drop is a core feature here. The position can also be set in the *widget*'s properties. When the *TableLayoutPanel* is used, a *widget*'s *Dock* property has to be set to „None”, in order to move it to another cell in the panel. The *Toolbox* is divided into categories, like *Common Controls* for buttons etc. and *Containers*, which can be faded out.

Widgets can again be selected by a left click, will then be highlighted and provide controls to change their size. Selecting a *widget* enables to access its properties.

The properties are key-value pairs. Most values are strings. Some provide a drop-down menu, like the color properties, others, for example the *Dock* property, a simple dialog to choose a direction. Like the *Toolbox*, properties are divided into categories.

Events are handled like properties. In the *Events* tab the name of the event handling method is assigned to an event which shall be handled on that *widget*.

3.3.3 Advanced Features

Besides the selection of *widgets* in the *Document Outline* or directly in the design view, they can also be selected in the context menu.

Multiple *widgets* can be selected at once, either by pressing *Control* while selecting the *widgets* with the mouse or by drawing a frame around these. Different values for properties are hidden here, but can also be changed for the selected *widgets* together. With the tool bar it is also possible to make selected *widgets* the same height, width or size. A selected *widget* can be changed in size and position by direct manipulation, too.

Furthermore, a row's or column's height or width can be adjusted by directly manipulating the border lines of a *TableLayoutPanel*.

Visual Studio provides copy, cut, paste and undo functionality.

3.3.4 Conclusion

When we are planning to design a *GUI* with a complex layout in Visual Studio, we basically have two options: the *TableLayoutPanel* or the default layout. The former presumes the usage of the *Column and Row Styles Dialog*, as direct manipulation of rows or columns is often not exact enough. One problem with this dialog is that it provides configuration options from a layout's and not a *widget*'s perspective. Hence, it lacks layout properties, like a *widget*'s position in the grid or the number of cells it covers and one often has to switch between this dialog and the *Properties*. Furthermore, the configuration is rather done on a textual than on a visual basis. This leads to a less intuitive usability compared to the *GridBagLayout Customizer* in NetBeans, although the same results can basically be achieved. Again we think the usability could improve with the integration of the dialog into the main view.

The default layout provides an easy and fast solution. The tool bar allows a direct access to its configuration and it does not share the problems we experienced with the *Free Design* in NetBeans. It probably is the better choice for a layout in Visual Studio.

The basic features, like adding *widgets*, accessing properties and event handling, are handled similar to NetBeans; thus, these are as expected.

As an alternative to the hierarchy, *widgets* can be selected in a context menu. However, as it is more clearly and also provides additional functionality, we think the hierarchy is the better solution.

An easy way to add user defined *widgets* does not exist in Visual Studio. Although it is likely to be possible somehow, we have not been able to figure that out. Anyway, as we are particularly interested in features which are easy to use, we have canceled our examination on this one after we invested a reasonable amount of time.

Finally, when we disabled some visual effects on our system and used the default layout, the *WYSIWYG* principle applied. Otherwise the final window's look was slightly different from that of the design (the application uses the system's native *look and feel*, but the design does not, at least not in every way, that's probably why the system configuration matters here).

3.4 GUI Builder

When running the GUI Builder, the language (and version), the application shall generate code for, has to be chosen. This choice does however not influence the look or functionality of the program. The difference is in the *widgets* which can be used. When *Tcl/Tk* (8.4) has been chosen, the main application starts:

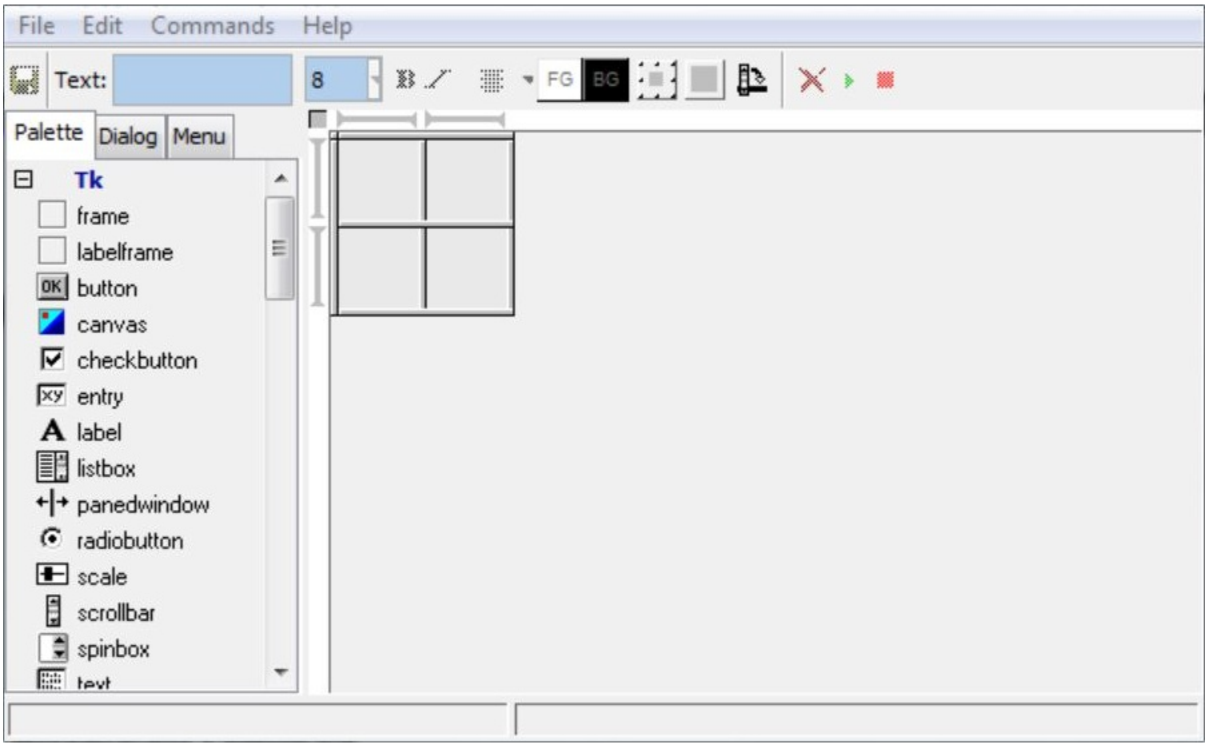

Illustration 13: Designing a main window in the GUI Builder

Located at the top are the main menu and the tool bar. On the left are the *Palette*, *Dialog* and *Menu* tabs and the design view on the right.

The tool bar provides direct access to some of the most important properties of a *widget*.

As before, the *Palette* contains a list of *widgets* which can be added to the design. The *Dialog* tab is a hierarchy view and the *Menu* tab is related to menu design.

The design view has some controls attached to it, see 3.4.1 for a description.

3.4.1 Layout

There is a single type of layout in the GUI Builder: the *Grid* layout from *Tcl/Tk*. Referring to 2.3.2, this is a structure based on a grid with flexible row and column sizes and numbers. A row's or column's size, as well as the weight and a padding for every *widget* in that row or column, can be adjusted in a dialog. To do so, a row, a column or both are selected via the corresponding control attached to the design view by a left click and the dialog is accessed in the *Edit* menu. They can also be adjusted in a single dialog, but each separately. Such a dialog provides a simple view on a textual basis, as illustrated in Illustration 14. Rows and columns can be added to or deleted from the grid via the *Edit* menu.

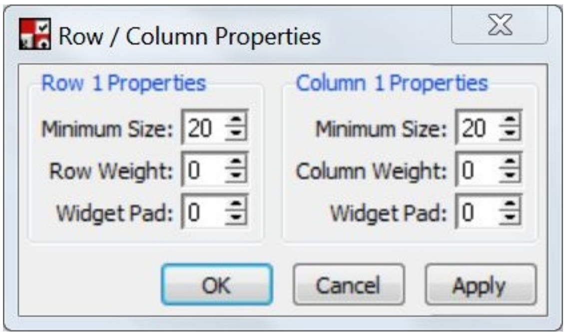

Illustration 14: Customizing Rows and Columns

Additional layout properties, like another padding or the *sticky* option to define the position in the cell, are related to the *widgets*. Note that a *widget*'s *rowspan* or *columnspan* property can solely be defined by direct manipulation, i.e., by selecting the *widget* and dragging one side to another cell. There is no other way to achieve this; the *Properties* dialog (see 3.4.2) lacks this option.

3.4.2 Core Features

A *widget* can, as before, be added to the design by dragging it from the *Palette* to the desired cell in the grid. Thus, drag and drop is again a core feature, as there is no other way to add *widgets*.

By left clicking a *widget*, a row control or a column control, it is selected. The selected *widget* or control is highlighted. Selecting a *widget* also attaches controls to it to define its span properties, as mentioned above. A selected element's properties can be accessed in the *Edit* menu. A *widget*'s *Properties* dialog can also be shown by a double click on the *widget*. As usual, this dialog provides a list of key-value pairs, where a string value or a choice from a drop-down menu is assigned to a property. In this context the *sticky* option is an exception. Its value is defined in a separate dialog, where the *widget*'s position in its cell(s) is chosen, similar to some properties in Visual Studio.

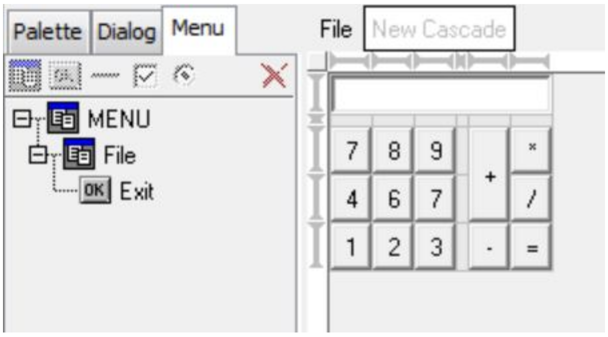

Illustration 15: Designing a menu and the final example GUI-design

Designing a menu works a bit different than usual. In the *Menu* tab the window's menu is represented by a tree. Each node corresponds to a menu on the menu bar and a leaf to an item in the parental menu. A menu can be added via the „New Cascade" label shown in Illustration 15. Clicking this adds a menu with a generic name. An item is inserted by one of the buttons on the top of the *Menu* tab. This is either a button (a common menu item), a separator line, a check box or a radio button item. Further configuration of menus and items can be performed in the *Properties* dialog, like normal *widgets*, though it can solely be accessed by a double click here.

Via its properties, an event handler can also be assigned to a *widget*, but with a restriction: A handler can only be assigned to the *command* option, which corresponds to the *widget*'s default event. I.e., on a button for example, it can be defined what happens on a left click, but not on a right click. The event handler can either be the name of the procedure to call or a short statement, which directly defines the event handling code.

3.4.3 Advanced Features

With the button in the upper left corner of the design view it is possible to toggle an alternative drawing of the grid's borders. This switches between the thick lines shown in Illustration 13 and the thinner shown in the other screenshots. When the cursor is above one of the grid borders, it changes to an arrow. The thicker lines make this easier to achieve. The borders can then be dragged to adjust row or column heights or widths.

Besides adding rows or columns to the design via the menu, it is also possible to drag a *widget* from the *Palette* to the empty space outside the grid, in order to extend it.

Furthermore, a selected row's or column's control can be clicked again to switch between a *Row* or *Column Weight* property of one and zero.

The tool bar contains controls to directly access some of a *widget*'s most important properties, an option to save the current design and options to run or stop the preview mode.

Finally, the *Dialog* tab provides the same functionality as directly selecting or double clicking a *widget* in the design view and in fact also selects the *widget* there. This hierarchy can also be traversed with options like *Next Widget* and *Select Parent* in the *Commands* menu.

3.4.4 Conclusion

With the *Grid* layout, the GUI Builder offers the flexibility to create a *GUI* with a simple or a complex layout. It is quite similar to the *GridBagLayout* and the *TableLayoutPanel*. However, the implementation has some drawbacks. The layout has to be configured at different places. It would be preferable to perform all of the corresponding modifications in the main view, in order to get a better overview. Furthermore, the tool bar mixes common and layout properties up. There are also just some of the properties accessible. The overall usability would probably advance, if an intuitive to use customizer, similar to the one for the *GridBagLayout*, would be integrated into the main view.

In addition, the menu design differs from the design of common *widgets*. There is only one *widget* selectable at once and the application lacks an undo function, the possibility to design custom *widgets* and to bind a handler to any type of event. An option to nest layouts, for example by using a container *widget*, like a *frame*, is also missing. It is indeed possible to add a *frame* to the design, but it can take a single *widget* only. Together with the missing custom *widget* option, this is a weighty drawback.

Nevertheless, the GUI Builder is overall easy to use and provides most of the options also available in the builders examined before. The results are as expected. Illustration 15 shows the final design of our example *GUI* and Illustration 16 the running preview, which is identical to the final application (except for the GUI Builder logo).

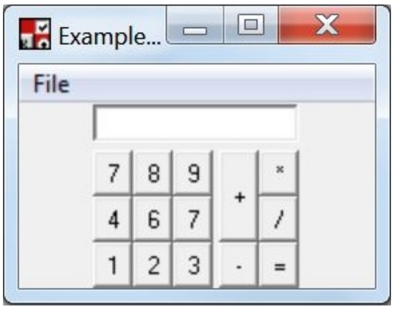

Illustration 16: The final example GUI

4 Requirements and Restrictions

4.1 *Requirements*

From our analysis of *GUI-builders*, we know which features a builder must have, as well as which comfort features it should provide to achieve a better usability. Thus, we are able to derive requirements for our own builder. As usual there are non-functional and functional requirements. The non-functional ones describe general aspects regarding the quality the application has to consider and the functional ones cover the concrete features or functions of the software [SWEBOK 2004].

4.1.1 Non-functional Requirements

- **WYSIWYG**. The key to a well usable interface is the user's expectations. Any action of the user should lead to a comprehensible result in the *GUI*-design, i.e., as he expects. The interface should then be intuitive to use. But as different users do not have the same expectations, this requirement also includes to provide different ways to achieve the same result where appropriate. Furthermore, the design should produce exactly the appearance in the final application the user has created. However, the design view may be decorated with context sensitive controls to achieve the usability aspect in the first place.
- **Downward Compatibility**. The requirements will make it necessary to extend *Curry*'s *UI*-library (see below). As there are already some extensive applications based on this library, we have to respect downward compatibility when implementing the extensions, in order to avoid introducing errors to the former.

4.1.2 Functional Requirements

Referring to what we have found out about the priorities of features, we classify the functional requirements into three classes:

1. **Major**: These are the core features, i.e., the must-have features which are necessary to create a *GUI*.
2. **Minor**: These requirements are basically derived from the advanced features. They cover useful features we have examined, as well as features we think a user expects a *GUI-builder* to provide.
3. **Optional**: The optional requirements will also form and have been derived from advanced features, but are relatively unimportant.

Note that these are general requirements, which are applicable to any *GUI-builder*. Due to implementation reasons and restrictions, some requirements may belong to another class they have been put into or we may even have to entirely deviate from others. Nevertheless, the goal is to completely cover at least the major and some of the minor ones in the first instance.

Major

- The software provides some kind of *layout manager*.
- It is possible to configure any of the layout's properties in the main view.
- A *widget* can be chosen from a list and inserted into the design.

- A *widget* can be removed from the design.
- A *widget* can be selected with the mouse. This highlights the *widget*.
- Any of a selected *widget*'s properties can be configured in the main view.
- The properties are represented by a complete list of key-value pairs. A value can either be typed in or chosen from a list.
- Configuration changes lead to corresponding changes of the design.
- An event handler's callback function is a *widget*'s property. It can be bound to any kind of event.
- A menu can be added to or removed from the design.
- A menu can be configured like any other *widget*.

Minor

- Where possible, controls can be used instead of or in addition to pure textual values for layout configuration.
- A *widget* can be dragged from a list to the design.
- A *widget* can be moved in the design by drag and drop.
- Multiple *widgets* can be selected at once.
- When multiple *widgets* have been selected, they can be configured at once.
- The application either provides a way to nest layouts or to define and use custom *widgets*.
- *Widgets* can be accessed via a tree-like hierarchy.
- A *widget* can be copied, cut and pasted.
- Multiple *widgets* can be copied, cut and pasted at once.
- Any action can be undone.

Optional

- Depending on the amount of *widgets*, they are divided into categories.
- Depending on the amount of properties a *widget* can have, the properties are divided into categories.
- A selected *widget* provides controls to directly manipulate its size. This requirement is optional, as, Depending on the layout, it is possibly easier for the user to configure layout or *widget* properties. Manipulation on a textual basis is also more accurate.
- The layout's configuration can directly be manipulated. Due to the same argument as above, this is an optional requirement.
- A hierarchy grants access to layout properties. This can also be a major requirement, if there is no other way to access layout properties.
- *Widget* properties are tailored to the concrete *widget*.
- The software provides a system to aid the user with the setup of event handlers.

4.2 Restrictions

Curry's *UI*- and *GUI*-library evolved since they have been announced the first time. But they still lack some features we would require to realize any of the requirements in every way. We provide implementations for some in the next chapter, but others would be beyond the scope of this work. Also, due to our downward compatibility requirement, we can extend the library, but have to avoid changes to existing functions where possible. Otherwise we had to be extremely careful, which is a high risk and would require extensive testing. Thus, the following features will not be supported, but we will provide a workaround where possible.

- So far, there is just one kind of *layout manager* in *Curry*: structures (*Row*, *Column*, *Matrix*) based on the *Grid* layout. Hence, there is no option to place *widgets* pixel-wise. Therefore we can't provide a visual (animated) drag and drop feature without implementing such an absolute layout, nor does the library.
- *Tcl/Tk* and thus the *UI*-library do not support transparent *widgets*. In addition, there is no absolute layout, as stated above. Hence, our *UI-builder* cannot draw controls on top of other (selected) *widgets*. Consequently, there are drawbacks on direct manipulation.
- The event handling on menus is limited. The default event (left mouse button) opens the menu and cannot be deactivated, but another handler for the same event can be assigned. Event handlers can also be assigned to menu items, but just for the default event. Due to the implementation of menus and especially their content, the configuration will be different than that of other *widgets*. Furthermore, it is also not possible to define or use context-menus.
- Attaching scroll bars is limited to only a few kinds of *widgets* and they are especially not available to layout *widgets*. Hence, screen space is limited, too.

5 Extensions

Curry's *UI*-library is lacking some features we will require to create a *UI-builder*. Thus, before we can start with the actual design and implementation, we have to extend the library with these and also a few other features, we think are important.

The library basically consists of two parts: on the one hand, a low-level part with either the module `GUI` for a desktop application based on *Tcl/Tk* or the module `HTML` for a *WUI* and, on the other hand, the module `UI` and its corresponding modules for a more abstract approach. Therefore, we also split the changes into two groups: The changes of the modules `GUI` and `UI2HTML` (implementing the logic for a *WUI*) basically are about functionality. The goal of the extensions of `UI` and the corresponding module `UI2GUI` is usability.

5.1 GUI

The following section describes the changes and extensions of the module `GUI`.

5.1.1 Rowspan and Columnspan

A quite important property for components laid out on a grid- or table-like structure is the possibility to span multiple columns or rows. Otherwise, the component would be limited to a single cell and thus different sized *widgets* would require the user at least to set absolute sizes. Consequently, this would limit the benefit of a *layout manager*. As these options do not exist in the module `GUI` yet, we implement them.

We extend the `ConfItem` data type for the new properties:

```
data ConfItem = … | ColSpan Int | RowSpan Int
```

An item's argument defines the number of columns or rows to span. We also extend the function `config2tcl`, which is responsible for the generation of a *Tcl*-string from a `ConfItem`, here for example for `RowSpan`, using the *Tk*-option `rowspan`:

```
config2tcl _ _ label (RowSpan rows) =
    "grid configure " ++ label ++ " -rowspan " ++ show rows ++ "\n"
```

The new properties can now be employed as any other, as the example below shows. `ColSpan` and `RowSpan` just increase a *widget*'s display area, so one may have to assign additional properties like `Fill` and `Anchor` to make use of the additional space.

```
widget = Matrix [] [
        [
            PlainButton [Text "ColSpan2", ColSpan 2]
        ],
        [
            PlainButton [Text "Button1"],
            PlainButton [Text "RowSpan2", RowSpan 2]
        ],
        [
            PlainButton [Text "Button2"]
        ]
    ]
```

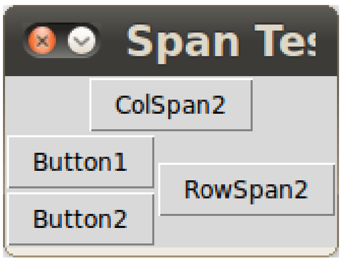

Illustration 17: ColSpan and RowSpan

5.1.2 Empty Cells

A new problem arises with the implementation of `RowSpan` and `ColSpan`: *widgets* can overlap each other, as in Illustration 18, where a *widget* spanning two columns is overlapped by another one in the same row.

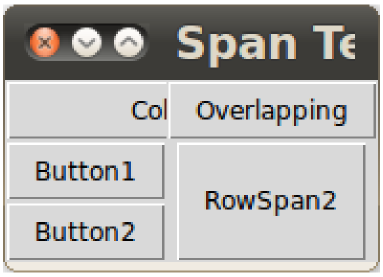

Illustration 18: Overlapping of a spanning widget

A solution in *Tcl/Tk* would be to leave a cell empty, i.e., to increase the column index of the button labeled "Overlapping". This is not an option in the *GUI*-library of *Curry*, as we do not directly define indices, but let the library generate them by the order *widgets* have been defined in a layout. Because these *widgets* are passed as a list to the layout, leaving elements of the list empty is also not an option, as this would produce a compiler error.

We define a new *widget* kind to solve this problem: the `NULL` *widget*. The `NULL` *widget* is a symbolic one, i.e., does only exist in the *Curry*-definition of the *GUI*, but does not have an own representation in the underlying *Tcl/Tk*-program. We extend the `Widget`

```
data Widget = … | NULL [ConfItem]
```

and again the `ConfItem` data type with new constructors:

```
data ConfItem = … | NoGrid
```

The `NoGrid` property marks a *widget* symbolic. It is required for a `NULL` *widget* to be in fact symbolic in the function `widget2tcl`, where we extend the rules for the layout *widgets*. We exemplary show this for a `Row` here:

```
1  widget2tcl wp label (Row confs ws) =
2    ((if label=="" then "wm resizable . " ++ resizeBehavior wsGridInfo++"\n"
3      else "frame "++label++"\n") ++
4     wstcl ++
5     (snd $ foldl (\ (n,g) l->(n+1, if elem NoGrid l then g else
6                                g++"grid "++label ++ labelIndex2string (96+n)
7                                ++" -row 1 -column "++show n++" "
8                                ++confCollection2tcl confs
9                                ++gridInfo2tcl n label "col" l ++ "\n"))
10                 (1,"")
11                 wsGridInfo),
12    wsevs)
13   where (wstcl,wsevs) = widgets2tcl wp label 97 ws
14         wsGridInfo = widgets2gridinfo ws
```

The code translates a *widget* to a *Tcl*-string by parsing its sub-*widgets* and their event handlers via `widgets2tcl` (lines 4 & 13 – note the additional „s") and mapping them to the grid (lines 5-10). Note the "1" in line 10: As a consequence indexing *widgets* in the grid always starts with one (and not zero, as one may be used to) and we have to take this into account anywhere we use these indices. Details are not important here, but that the concrete *widget* is not available when it comes to "gridding" is a problem. Therefore, a test whether the current *widget* to grid is a `NULL` *widget* is not possible without many changes (and would possibly break downward compatibility). But the *widget*'s `ConfItems` are available (in `l`) and we can define that a *widget* with `NoGrid` shall not be mapped to the grid and just return the previously created *Tcl*-string. Hence, the only change is in line 5 and written in bold letters.

We also have to implement a `widget2tcl` pattern for the `NULL` *widget* itself, but this is very simple, as it just returns the results of `config2tcl`. The implementation of `config2tcl` for `NoGrid` is also trivial, as it is just the empty string. We could actually return this string and an empty list of event handlers directly, but we will see below that a `NULL` *widget* may also have a reference and therefore a call of `config2tcl` is required.

Note that `NoGrid` is similar, but not identical to `Display`, because a *widget* must already exist in the grid for the latter, but not the former. `NoGrid` can, of course, also be used for other *widgets* than `NULL`.

Finally, to ensure that a `NULL` *widget* always has the `NoGrid` property, we extend the function `getConfs` for this kind of *widget* to add the property, if it is not already in the list:

```
getConfs (NULL confs) = if elem NoGrid confs then confs else
                                 (NoGrid : confs)
```

Pattern matching, instead of a call of this function, could however undermine this mechanic, but so far this is not the case.

Illustration 19 shows the *GUI* from above with an additional `NULL` *widget* behind the one spanning columns. There is no *widget* overlapping another one anymore. This is the example's source code:

```
widget = Matrix [] [
          [
              PlainButton [Text "ColSpan2", ColSpan 2, FillX],
              NULL [],
              PlainButton [Text "Overlapping"]
          ],
          [
              PlainButton [Text "Button1"],
              PlainButton [Text "RowSpan2", RowSpan 2, FillY]
          ],
          [
              PlainButton [Text "Button2"]
          ]
      ]
```

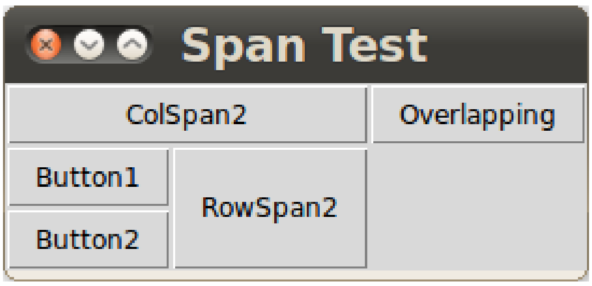

Illustration 19: RowSpan and ColSpan with an additional NULL widget

5.1.3 Images

Another very important property for a *widget* is the possibility to display images. These give the user the opportunity to design a *GUI* with an individual look and it's hard to imagine an advanced interface without icons (especially a *GUI-builder*). Images have not been implemented in the *GUI*-library yet; therefore, we are implementing them now.

In *Tcl/Tk* an image can be loaded by the command

```
image create photo <name> -file <path>
```

where `photo` denotes the type of the image to be a GIF-, PPM- or PGM-file and `<name>` is a handle for the loaded image. There are alternatives for `photo`, but a GIF seems to be sufficient here.

Consequently, an image is just defined by its path (we abandon options to configure, e.g., the image's height or width). We define a new `ConfItem` reflecting this, where its argument is the path:

```
data ConfItem = … | Image String
```

In *Tk* the image can be assigned to a *widget* by the option `-image <name>`. Therefore, we add a rule to `config2tcl` to create the image and assign it to a *widget* by performing the *Tcl*-commands. This is similar to 5.1.1. For the image's name we choose “image” plus the *widget*'s reference name (a *widget*'s reference name in this context is its *Tk*-reference, e.g., “.a.b.c”, where dots are replaced by underscores).

The following example shows the usage of `Image`:

```
widget = Row [] [
        PlainButton [Text "text disappears", Image “/test_image.gif"],
        CheckButton [Image "/test_image.gif"],
        Label [Image "/test_image.gif"]
    ]
```

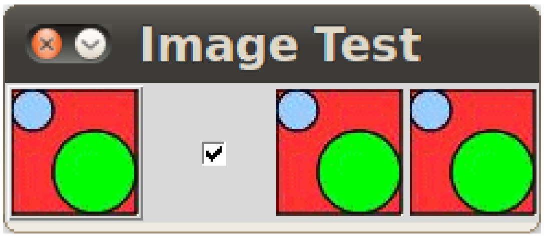

Illustration 20: Widgets with images. From left to right: button, check button and label

We assign a test image to a `PlainButton`, a `CheckButton` and a `Label`. `MenuButtons` and menu items (`MbuttonC`) also support this property. A side effect of the image option in *Tk* and thus the `Image ConfItem` is that it overrides a text label, as shown for the button. Scaling a *widget* with an associated image by using `Width` and `Height` modifies the *widget*, but not the image.

5.1.4 MatrixC

As stated above, a `NULL` *widget* may or even must have a reference. This is the case for any *widget* and especially for the layout *widgets*, as we will see in the next section. A reference `WRef` is of the type `ConfItem`, but a `Row`, `Col` or `Matrix` cannot have any `ConfItems`. Fortunately, there have already versions of `Row` and `Col` been implemented eliminating this deficit, namely `RowC` and `ColC`. Though, this is not the case for the matrix; thus, we are doing this now.

We extend the data type `Widget` again:

```
data Widget = … | MatrixC [ConfCollection] [ConfItem] [[Widget]]
```

As we have seen above, we also have to extend `widget2tcl` when we are creating a new kind of *widget*. We can combine the implementations of the common `Matrix` and those of `ColC` or `RowC`. Consequently, the resulting code is very similar to that shown above and nearly the same as for the common `Matrix`. We just parse the `ConfItems` and compute event handlers, add the former to the output string and the latter to the ones from the matrix' sub-*widgets*.

In addition, we have to extend the functions `getConfs` and `propagateFillInfo` (for any new kind of *widget*), but this is trivial and therefore we do not show the details here.

5.1.5 Insert, Delete, Move and Configure

In order to implement a *GUI-builder*, it must be possible to add and remove *widgets* to or from a design at runtime, as well as moving and configuring an existing *widget*. One can basically choose out of two approaches: either using *live widgets*, i.e., concrete *widgets* or proxies for the actual *widgets*, like images of these. The latter is likely to be easier to configure and could probably be realized with a canvas-*widget* and thus with the existing technique for the runtime configuration of *widgets* (see below). But a weighty disadvantage is that these proxies also had to emulate the semantics, especially regarding the layout of the *widgets* they are representing, in order to apply the *WYSIWYG* principle. Therefore, proxies would require some additional effort to realize. Consequently, we stick to the *live widgets*. As it is not yet possible to dynamically insert, delete or move them, we are implementing these features now.

So far, the module `GUI` just supports the dynamic configuration of *widgets* with `ConfItems` at runtime. To achieve this, the implementation defines a data type

```
data ReconfigureItem = WidgetConf WidgetRef ConfItem
```

To reconfigure an existing *widget*, one creates such a `WidgetConf` with the reference of the *widget* to configure and a new `ConfItem`. After that, the resulting `ReconfigureItem` can be returned by the current event handler to execute the change. This approach, where a command is encapsulated into a data object, can be seen as a variant of the *command pattern*, known from object oriented languages. Referring to 2.1 this is very useful; hence, we adapt it for the insert, delete and move features. We could stick to a `WidgetConf` for a simple reconfiguration of a *widget*, but to cover the *command pattern* in every way, especially regarding the required information to undo configurations, we also develop a new constructor `WidgetConfigure` (we keep `WidgetConf` for downward compatibility).

We extend the data type:

```
data ReconfigureItem =
 …
 | WidgetInsert Widget Widget (Maybe Int) (Maybe Int) (Maybe Bool)
 | WidgetDelete Widget WidgetRef
 | WidgetMove Widget WidgetRef (Maybe Int) (Maybe Int) (Maybe Bool)
 | WidgetConfigure Widget WidgetRef ConfItem Bool
```

For a `WidgetInsert` we require

1. the parent of the *widget* to insert, in order to derive the new *widget*'s reference and update existing children (see below)
2. the new *widget* itself
3. its row position, if the parent is a `RowC` or `MatrixC`
4. its column position, if the parent is a `ColC` or `MatrixC`
5. whether the new *widget* should be inserted into a new or an existing row, if the parent is a `MatrixC`

and the required parameters for a `WidgetDelete` are

1. the parent of the *widget* to delete, in order to update other children
2. the *widget*'s reference

For a `WidgetMove`

1. the parent of the *widget* to move, in order to update the children

2. the *widget*'s reference
3. the new row position (see `WidgetInsert`)
4. the new column position (see `WidgetInsert`)
5. new or existing row (see `WidgetInsert`)

are necessary and a `WidgetConfigure` requires

1. the *widget*, to keep track of its configuration before the change
2. the *widget*'s reference
3. the new `ConfItem`
4. whether the configuration for that item should be deleted, i.e., reset to a default value

After the event handling code with the reconfiguration terminated, the scheduler takes over control again. It checks if there are `ReconfigureItems` pending and performs the reconfiguration in the function `configAndProceedScheduler`. There we have to insert the code, which performs the actual insertion, deletion, movement or configuration**.**

For a `WidgetInsert` we just call a function corresponding to the given arguments, e.g., `insertIntoMatrix` if row and column positions, as well as the boolean for a new row, are of the type `Just`. For a `WidgetDelete` we also have to retrieve the actual *widget* from the parent, at first.

The functions `insertIntoCol`, `insertIntoRow` and `insertIntoMatrix` are quite similar; hence, we confine ourselves to show the latter, which is also the most complex one:

```
  insertIntoMatrix :: GuiPort -> Widget -> Widget -> Int -> Int -> Bool ->
    IO ()

1 insertIntoMatrix gp (MatrixC confs _ ws) child r c newRow = send2tk tcl gp
2     where   label           = getLabel $ getConfs child
3             (_, bottom)     = splitAt (r-1) ws
4             (_, right)      = splitAt (c-1) (if null bottom
5                 then [] else head bottom)
6             ws2grid         = if newRow
7                 then matrix2grid ([child] : bottom) r c confs
8                 else matrix2grid [child : right] r c confs
9             (wtcl,_)        = widget2tcl gp label child
10            tcl             = wtcl ++ ws2grid
```

The function's goal is to create a *Tcl*-string for the *widget* to insert, similar to `widget2tcl` and send it to *Tk* (line 1). `Widget2tcl` can only generate a *widget*'s label (and reference) when it parses the whole *GUI*, starting with the root *widget*, but we just need a label for a single *widget* here, which may be located anywhere in the hierarchy. Furthermore, `widget2tcl` does not perform the mapping to the *Grid* when it is called with a non-layout *widget*. Thus, a call of `widget2tcl` alone is not sufficient. See below for the actual binding of the label to the *widget*'s free variable for its reference. We just retrieve the label from it here (line 2 – consequently the *widget*'s reference must already be bound here). In addition, in order to prevent errors, we also would like a *widget*'s position to always be consistent with its index. Hence, we have to update existing *widgets* in the parent (lines 6-8), besides the parsing of the new one (lines 9 & 10). In order to get a list of *widgets* to update, we split the matrix' children at the given row and column positions (`splitAt` starts with index zero). If the new *widget* shall be inserted into a new row, we have to (re-)grid it and the successive rows (line 7), otherwise just the new *widget* and those in successive columns (line 8). We are then performing the creation of the *Grid* string in

```
  matrix2grid :: [[Widget]] -> Int -> Int -> [ConfCollection] -> String

  matrix2grid [] _ _ _ = ""

1 matrix2grid ([] : rows) r _ confcolls
2     | not $ null rows   = matrix2grid rows (r+1) 1 confcolls
3     | otherwise         = ""
4
5 matrix2grid ((w : ws) : rows) r c confcolls = if elem NoGrid (getConfs w)
6     then matrix2grid (ws : rows) r (c+1) confcolls
7     else "grid " ++ label
8         ++ " -row " ++ show r ++ " -column " ++ show c ++ " "
9         ++ confCollection2tcl confcolls
10        ++ gridInfo2tcl c label "col" wsGridInfo ++ "\n"
11        ++ matrix2grid (ws : rows) r (c+1) confcolls
12    where   label            = getLabel $ getConfs w
13            (wsGridInfo : _) = widgets2gridinfo [w]
```

This function maps the new *widget* to the *Grid* and moves successive *widgets*. This is basically the same as in `widget2tcl`, but without parsing the *widgets* themselves. We create the grid string on the basis of a *widget*'s grid information, derived from its `ConfItems` (lines 10 & 13) and the current row and column indices (line 8). When we have finished this, we increase the column index (line 6 or 11) and when a whole row has been processed, we increase the row index (line 2). Of course, we also have to respect the `NoGrid` property here.

Deleting a *widget* with

```
deleteWidget :: GuiPort -> Widget -> Widget -> IO ()
```

works very similar; thus, we do not go into details here. We retrieve the *widget*'s position in its parent. If the *widget* does not have the `NoGrid` property, we are then deleting it in *Tk*, resp. creating the corresponding string. Finally, we update the remaining *widgets*, i.e., move them leftwards or upwards, where we reuse `row2grid`, `col2grid` or `matrix2grid` and send the resulting string to *Tk*.

There are even more similarities between inserting and moving a *widget*. With, e.g.,

```
moveInRow :: GuiPort -> Widget -> WidgetRef -> Int -> IO ()
```

we remove a *widget* from its current position in its parent (here: `Row`) and insert it at the new one. To keep it simple, we just update any child by the functions `row2grid`, `col2grid` or `matrix2grid` this time and do not compute which concrete *widgets* have to be re-gridded.

Finally, in order to configure a *widget*'s `ConfItems`, we create the function

```
configureWidget :: GuiPort -> WidgetRef -> ConfItem -> Bool -> IO ()
```

which is a combination of `config2tcl` and `delconfig2tcl`. The latter is a new function we use to delete a `ConfItem`, if the corresponding argument of `configureWidget` is `True` or more precise, the function sets a *widget*'s property to a default value. This also covers event handlers, where we define the handler to do nothing. For example

```
delconfig2tcl _ label (ColSpan _) = "grid configure " ++ label
    ++ " -colspan 1\n"
```

defines the *widget* with the given label to span a single column.

A disadvantage of the implementation of `configureWidget` is that a `ConfItem` with a concrete value is required to determine the item's type, even if it shall be deleted. But the value is arbitrary and therefore this should not be a serious problem.

Now that we have inserted, deleted, moved or configured the *widget*, there are still the event handlers to add or remove to or from the schedule. After reconfiguring *widgets*, `configAndProceedScheduler` calls `configEventHandlers`, which is responsible for this task. We retrieve a *widget*'s (`child`) and any of its children's event handlers by `widgets2handler` and for, e.g., a `WidgetInsert`, just add them to a list:

```
configEventHandlers evs (WidgetInsert _ child _ _ _ : confitems) =
  configEventHandlers (widgets2handler child ++ evs) confitems
```

For a `WidgetDelete` we have to retrieve a *widget* via its reference from its parent and filter the handlers out of the list with `filterHandlers`:

```
configEventHandlers evs (WidgetDelete parent ref : confitems) =
  let   child       = case getWidgetByRef parent ref of
             (Just w)    -> w
             Nothing     -> error ("The widget is not a child of the “
                 ++ “given parent!" )
        handlers    = widgets2handler child
  in configEventHandlers (filterHandlers handlers evs) confitems
```

For a `WidgetConfigure`, if the `ConfItem` is a `Handler`, we have to differentiate between an addition and a deletion of that item. We then just add or remove the handler to or from the list of handlers.

We do not show the details of `widgets2handler` or `filterHandlers` here, but we have to be careful when comparing handlers, as comparing the concrete functions may lead to an endless loop (and we had in fact some problems before figuring that out). Therefore, we can compare labels and event types only.

One may now wonder how these `ReconfigureItems` we just created can be used in a user's program. This is actually not so easy, as once the main *widget* has been defined and `runGUI` been called, the *GUI*-definition and hence the definition of the parent or the original *widget*, which are required for the `ReconfigureItems`, are lost. To solve this problem we could either reproduce *widgets* from the information we are receiving from *Tcl/Tk* or save the *GUI*'s state somehow. We have examined the former approach and arrived at the conclusion that it would be quite difficult to implement, as, for example, the communication with *Tcl/Tk* is problematic, because the scheduler is blocking it while listening for events. Also, we (or the user) have to keep track of the *GUI*-structure and the *widgets*' references for these operations anyway. Thus, we take the latter approach, although it is a little complicated for the user at the first glance, but takes the *GUI* to a higher level, which provides more control.

To store the *GUI*-definition, the user can save it in an IO-reference and introduce it to event handlers. But how does the definition change on a `WidgetInsert`, `WidgetDelete`, `WidgetMove` or `WidgetConfigure`? We should not expect from the user to have knowledge about implementation details of the module `GUI`. Therefore, we define functions to create the `ReconfigureItems` and also an updated description. To keep it simple here in the first place (note that functionality is the goal, as stated at the beginning of this chapter), we just update the parent of the concerned *widget*.

We define a new function

```
createInsert :: GuiPort -> Widget -> Widget -> Maybe Int -> Maybe Int ->
    Maybe Bool -> (Widget, ReconfigureItem)
```

where the arguments are the same as for a `WidgetInsert`. Its result is the updated parent, as well as the `ReconfigureItem` itself. We exemplary examine the implementation for a `MatrixC` again:

```
1 createInsert gp parent@(MatrixC confs confitems ws)
2     child (Just r) (Just c) (Just newRow) =
3     (MatrixC confs confitems result,
4     WidgetInsert parent w (Just r) (Just c) (Just newRow))
5     where   parentLabel     = getLabel confitems
6             (top, bottom)   = splitAt (r-1) ws
7             (left, right)   = splitAt (c-1) (if null bottom
8                 then []
9                 else head bottom)
10            fillupCol       = replicate (r - 1 - length ws) []
11            fillupRow       = if newRow
12                then fillUp gp parentLabel (concat ws) (c - 1)
13                else fillUp gp parentLabel (concat ws)
14                    (c - 1 - (length left + length right))
15            label           = parentLabel
16                ++ nextlabel (concat ws ++ fillupRow)
17            mayberef        = getRef $ getConfs child
18            w               = case mayberef of
19                (Just ref)  -> bindRef gp ref label child
20                Nothing     -> child
21            result          = if newRow || null bottom
22                then top ++ fillupCol ++ (fillupRow ++ [w]):bottom
23                else top ++ fillupCol
24                    ++ (left ++ fillupRow ++ (w:right)):tail bottom
```

We already know most of the details from `insertIntoMatrix`, but have to redo them for the update, because it is not possible to gain results from the actual insertion or vice versa. But there are also some new aspects. One is the fill-up in the lines 10-14, resp. 22-24. We already stated above that indices should be consistent with the *widgets*' positions in a parent. To achieve this, we put in empty lists (rows) to fill the column part of the matrix up (lines 10 & 22 or 23) and `NULL` *widgets* into empty spaces in a row (lines 11 & 22 or 24 – remember that empty spaces in a list are not allowed). In `fillUp` we create the `NULL` *widgets* with references and bind them to labels derived from the parent's and those of already existing children. We do the same for the new *widget* (line 19). See the illustrations below for examples.

Labels are identical to those used in the underlying *Tcl/Tk*-program. Referring to 2.3.1, these are string values, like “.parent.child”, where “.parent” is the parent's label. In the module `GUI` a layout *widget*'s children get labels in the order they have been passed to their parent, represented by a character and maybe a number. Let “.b”, for example, be the parent's label. The children's labels will then be “.b.a”, “.b.b”, “.b.c”, … , “.b.z”, “.b.z1”, “.b.z2” and so on. To create a new label one may think that the number of children is sufficient information to achieve this. But firstly, due to deletion and insertion, this approach could lead to conflicting labels and secondly, it is absolutely possible that the parent does not contain *widgets* at all. Consequently, we require both, the parent's label and those of its children. Deriving these labels is a bit tricky to implement, but with the description it should be clear that we just have to find some kind of lexicographical maximum; thus, we can skip the details of the responsible function `nextlabel`. Note that the dependency on the parent's label, resp. its reference, is also the reason why it is only possible to insert into a `RowC`, `ColC` or `MatrixC` and that these must have bound references. Finally, to bind the label to a *widget*'s free reference, we implement the function `bindRef` using narrowing, similar to the common binding in `config2tcl`, but returning the *widget* instead of a string:

```
bindRef :: GuiPort -> WidgetRef -> String -> Widget -> Widget

bindRef gp ref label w
    | ref =:= WRefLabel gp (wLabel2Refname label) (showWType w) = w
```

We just call the constructor for the type `WidgetRef` and bind the result to the reference. Because we use the result of `bindRef` in `createInsert`, although we did not change the *widget*, we force the interpreter to evaluate this binding.

The following sketches sum up what happens on an insertion into the different layout types (an index in a box indicates that there is an element at that position and an empty box that there is nothing in it, resp. at that position in a *widget*'s list of children):

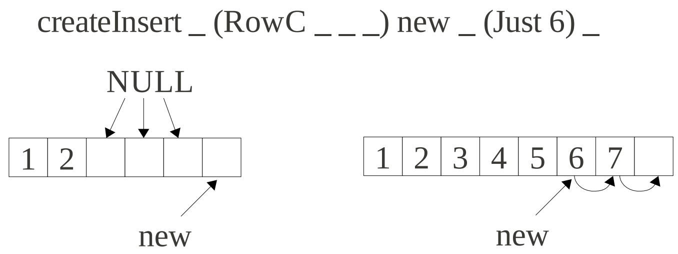

Illustration 21: Inserting into a RowC

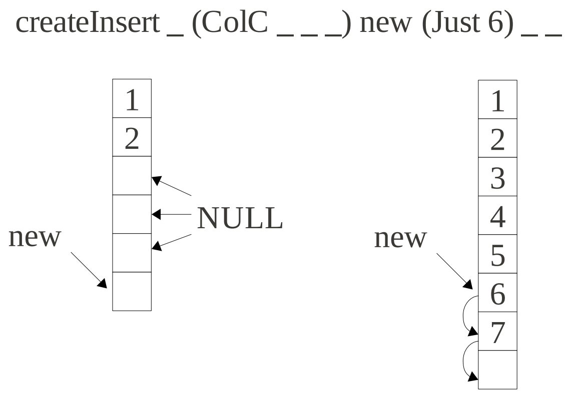

Illustration 22: Inserting into a ColC

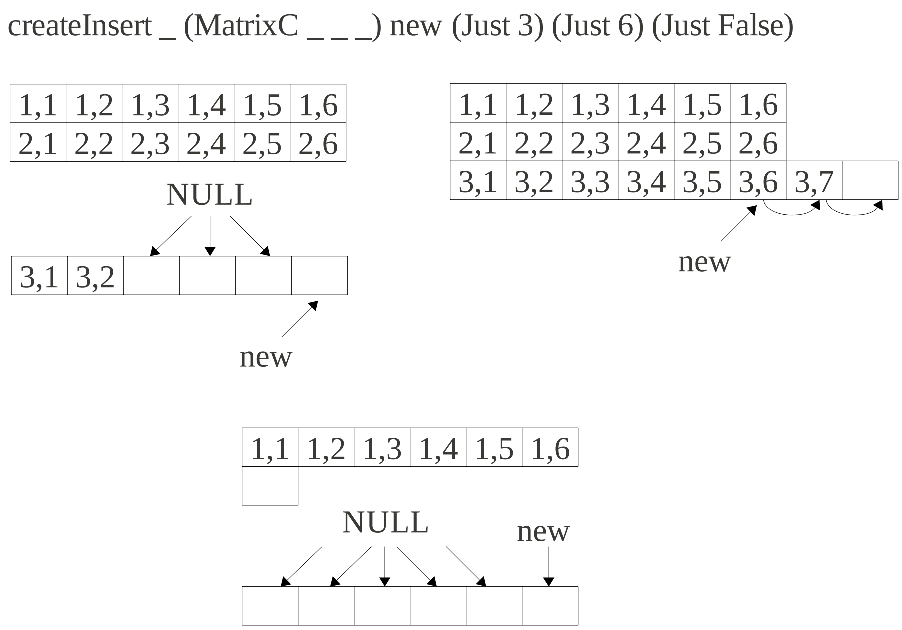

Illustration 23: Inserting into a MatrixC and into an existing row

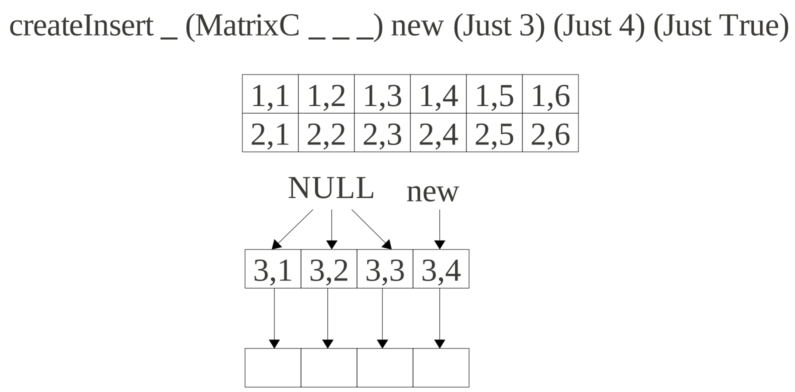

Illustration 24: Inserting into a MatrixC and into a new row

Creating a `WidgetDelete` by

```
createDelete :: Widget -> WidgetRef -> (Widget, ReconfigureItem)
```

is easier, as we do not have to perform any binding (and thus do not need the `GuiPort` here) or fill-up. We just remove the concerned *widget* from its parent.

We also implement the functions

```
createMove :: GuiPort -> Widget -> WidgetRef -> Maybe Int -> Maybe Int ->
    Maybe Bool -> (Widget, ReconfigureItem)
```

which is similar to `createInsert`, but without binding references and

```
createConfigure :: Widget -> ConfItem -> Bool -> (Widget, ReconfigureItem)
```

which adds or replaces a `ConfItem` to or in a *widget* – the latter if one of the same kind already exists – or delete the item if the boolean is `True`. We replace an event handler if the *widget* has one for the same kind of event, as more than one handler for the same event does not make any sense.

The following listing shows an example where we use the `createInsert` and `createDelete` functions:

```
import GUI
import IOExts

widget gui = do
    writeIORef gui mainWidget
    return gui
        where mainWidget = RowC [] [] [createButton 1 gui]

createButton n gui = PlainButton [WRef b, Text "Button",
        Handler DefaultEvent (leftHandler n gui),
        Handler MouseButton3 (rightHandler b gui)
    ]
    where b free

leftHandler n gui gp = do
    parent <- readIORef gui
    (newparent, ins) <- insertButton n parent gui gp
    writeIORef gui newparent
    return [ins]

insertButton n parent gui gp = return (newparent, ins)
    where   new                  = createButton (n+1) gui
            (newparent, ins)     =
                createInsert gp parent new Nothing (Just (n+1)) Nothing
```

```
rightHandler b gui _ = do
    parent <- readIORef gui
    (newparent, del) <- deleteButton b parent
    writeIORef gui newparent
    return [del]

deleteButton b parent = return (createDelete parent b)

main = do
    init <- newIORef (Row [] [])
    gui <- widget init
    w <- readIORef gui
    runGUI "Insert Delete Test" w
```

The *GUI* initially consists of a `RowC` and a single button. On a left click on the button, `leftHandler` adds another one behind the first and so on. A right click on the button deletes itself via `rightHandler`. Note that the `RowC` does not require a reference for this special case, as it becomes the main window with the label ".".

We use an IO-reference to save the main *widget* (we have to initialize it with some value of type `Widget` in the function `main`) and introduce its current value to event handlers. Due to this IO-reference, where the current state of the *GUI* has to be manually managed, the implementation of a simple application can become quite complex. If the *GUI* had nested layouts, we had to pattern match the whole *GUI*-structure up to the parent we actually want to change. Anyway, we achieved the functionality aspect of our goal and eliminate its deficits regarding usability in the next sections.

5.2 UI2HTML

The changes of `UI2HTML` are complementing the extensions of the library's low-level part. As our builder will be based on a *GUI*, we do not implement insert, delete, move or configure here. But the builder shall take advantage of the *UI*-library and the possibility to use a design for both, a *GUI* and a *WUI*. Hence, the new kinds of *widgets* and properties have to be implemented for a *WUI*, too.

In order to define a *WUI*, `UI2HTML` creates *HTML*-tags from *widgets*, where the attributes correspond to (`UI`-)styles. Layout *widgets* are mapped to `div` or `span` tags and an additional `table` tag. *JavaScript* implements the event handling. The latter is irrelevant here; thus, we stick to the mapping of *widgets* and styles.

5.2.1 Images

In `UI2HTML` a `UI.Style` is mapped to a *CSS*-style, resulting in an attribute for that element, like

```
style = “width: 100%”
```

for `Fill X`. For the new `Style Image` we simply add a mapping to the *CSS*-style

```
"background-image: url(“ ++ path ++ “); background-repeat: no-repeat;
    background-position: center”
```

where `path` is the `Style`'s argument, i.e., a URL. In *HTML*/*CSS* an image can be defined as a texture, i.e., the image is repeated until the available space is covered. As we just want a single image to be displayed on a *widget* (as for a *GUI*), we also add the `background-repeat: no-repeat` style. The following example shows how to use a button with an image in a *WUI*, as well as the differences between a *GUI* and a *WUI*:

```
widget = row [
        Widget Button (Just "Button") Nothing []
        [Class [Image "http://localhost/test_image.jpg"]] []
    ]
```

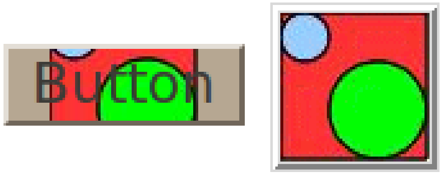

Illustration 25: Button with image in a WUI (left) and a GUI (right)

As shown in Illustration 25, the image defines the *widget*'s size and also hides the label in a *GUI*. In a *WUI* the image does not influence the *widget* at all.

5.2.2 Rowspan and Columnspan

`ColSpan` and `RowSpan` are a bit more difficult to implement, as they are different from the other styles. As stated above, a `UI.Style` is mapped to a *CSS*-style. But in *HTML* `rowspan` and `colspan` are attributes and not styles and furthermore not applied to the inner tags, but to the cells, i.e., the `td` tags in the parent's `table` tag. The cell, as well as the table itself, must also have "height: 100%" as a part of their style, otherwise `rowspan` would not have any effect. As the latter is constant, we can directly add it (see `table'` below). We have to define a translation for the cell attributes.

Common styles are mapped by `styleClasses2Attrs` to a pair of the type

```
(String, String)
```

which is the type of an attribute in the module `HTML`, resulting in

```
(“style“, val)
```

where `val` is the attribute's value. To differentiate between these and the span properties, we implement a function

```
styleClasses2CellAttrs :: [StyleClass] -> [(String, String)]
```

which just maps `ColSpan` to

```
(“colspan”, show n)
```

and `RowSpan` to

```
(“rowspan”, show n)
```

where `n` is the number of columns or rows to span. The function

```
widget2hexp :: Maybe (IORef State) -> Widget CgiRef (UIEnv -> IO ()) _ ->
    [HtmlExp]
```

computes the mapping of a *widget* to *HTML* in the module. Layout *widgets* are mapped to a `div` or a `span` tag with an additional `table` element. In order to access the table's cells, we replace, for example,

```
xs = case kind of
    Row -> [table' [map (\ w -> widget2hexp mbstateref w) widgets)]
```

where `xs` is a list of *HTML*-expressions for the elements of these `div` or `span` tags, with

```
xs = case kind of
    Row -> [table' [widgets2row mbstateref widgets]]
```

We map the `ColSpan` and `RowSpan` styles of the layout's children to cell attributes (or an empty list) in the new function `widgets2row` and pass them, together with the corresponding *HTML*-expression, to the function `table'`. In the customized version of this function, we are then creating an `HtmlStruct` for a `table` element, put the children into rows (`tr`) and cells (`td`) and combine each cell with a list of these attributes:

```
table' :: [[([HtmlExp],[(String, String)])]] -> HtmlExp

table' items = HtmlStruct "table" [("style","height: 100%;")]
    (map (\row->HtmlStruct "tr" []
        (map (\(item, attrs) -> HtmlStruct "td" (("style",
            "vertical-align: top; height: 100%;") : attrs) item) row))
        items)
```

We have highlighted the changes.

The implementation of `ColSpan` and `RowSpan` does not require `NULL` *widgets* here, as the interpreter automatically moves *widgets*, which would overlap the spanning one, aside. In fact, tests have shown that translating a `NULL` *widget* to, e.g., a `div` tag, would add empty space between a spanning *widget* and its successors. Therefore and to keep `UI2HTML` compatible with the high-level *UI*-implementation, we take `NULL` *widgets* into account, but just ignore them in `widgets2row`.

Illustration 26 shows the example from 5.1.2 in a *WUI*.

Illustration 26: RowSpan and ColSpan in a WUI

5.3 UI

On the abstract or high-level side of the *UI*-library we want to deal with aspects regarding the library's usability. But at first we complete it with the new kinds of *widgets* and properties.

As the module `UI` provides an abstract interface for a *UI*, the new elements do not implement any functionality here. Hence, they are realized by simple extensions of the existing data types:

```
data WidgetKind r act1 act2 = … | NULL | CommonListBox
    | ScrollBarH (Ref r) | ScrollBarV (Ref r)

data Style = … | ColSpan Int | RowSpan Int | LBList [String]
    | Image String
```

There has already been a `WidgetKind ListBox`, but as will be shown in 5.4 below, it is mapped to a `ListBoxScroll`, which is not a basic type, but a combination of a `Matrix`, a `ListBox` and a vertical, as well as a horizontal scroll bar. This quite complex mapping causes problems regarding reconfiguration. Therefore, we add the simpler `WidgetKind CommonListBox`, together with constructors for scroll bars as single *widgets*, which can now also be used for a `TextEdit` or a `Canvas` (scroll bars are, as `NULL` widgets, ignored in `UI2HTML`). A combination of these three and a `Matrix` behaves like a `ListBox` before, but each can be separately configured.

The `Style` `LBList` defines a `CommonListBox`' items. The meaning of the `Styles` `ColSpan`, `RowSpan` and `Image` should be obvious. Due to our implementation of the function `getConfs` in the module `GUI`, the `NoGrid` property is unnecessary for a `NULL` *widget* here.

The further additions are more advanced. In order to browse through a hierarchy of *widgets* or to find a special *widget*, we adapt some of the known functions from the modules `Prelude` and `List` for these purposes on lists to *widgets*. These will help to reduce the required input for delete and insert operations to the information which is really necessary.

We implement a function to map another one to a *widget* and any of its children:

```
mapUI :: (Widget a b c -> Widget a b c) -> Widget a b c -> Widget a b c
```

two functions to accumulate a *widget* and any of its children by applying a binary operator, either from top to bottom and left to right or vice versa:

```
foldlUI :: (d -> Widget a b c -> d) -> d -> Widget a b c -> d
```

```
foldrUI :: (Widget a b c -> d -> d) -> d -> Widget a b c -> d
```

one to find a *widget* satisfying a predicate:

```
findUI :: (Widget a b c -> Bool) -> Widget a b c -> Maybe (Widget a b c)
```

another one to find a *widget*'s index (again starting to count with one):

```
findIndexUI :: Ref a -> Widget a b c -> (Maybe Int, Maybe Int)
```

We furthermore implement a function to find a *widget* and replace it by an updated version, by comparing references and using `mapUI`:

```
updateUI :: Widget a b c -> Widget a b c -> Widget a b c
```

one to retrieve a *widget* by its reference, using `findUI`:

```
getWidgetByRef :: Ref a -> Widget a b c -> Maybe (Widget a b c)
```

another one to retrieve a *widget*'s parent:

```
getParent :: Ref a -> Widget a b c -> Maybe (Widget a b c)
```

and finally a function to retrieve a *widget* at a given position in its parent:

```
getWidgetAt :: Widget a b c -> Maybe Int -> Maybe Int ->
    Maybe (Widget a b c)
```

We exemplary analyze `findUI` here, as most of the others are quite similar:

```
1 findUI f w@(Widget kind _ _ _ _ ws) = if f w then Just w else
2     case kind of
3         Matrix wss  -> findUI' $ concat wss
4         _           -> findUI' ws
5     where   findUI' []          = Nothing
6             findUI' (w':ws')    = case findUI f w' of
7                 Nothing -> findUI' ws'
8                 justw   -> justw
```

In line 1, if the current *widget* satisfies the predicate, we are done. Otherwise, we have to differentiate between a `Matrix` and other *widgets*, as a `Matrix`' children are part of the `WidgetKind`. If this is the case, we can just concatenate rows and proceed with the resulting one-dimensional list. In the lines 5-8 we are then applying `findUI` to any child, until we have found the first occurrence of the desired *widget* or return `Nothing`, if there are no more children, indicating that the desired *widget* does not exist.

We implement these functions in the module `UI`, as they are useful for any *UI*, whether be it a *GUI*, *WUI* or something else. But actually we just use them for the *GUI*-part, as we did not implement insert, delete, move or configure for a *WUI*.

5.4 UI2GUI

The next step is to connect the modules `GUI` and `UI`. In contrast to `UI2HTML`, the responsible module `UI2GUI` implements almost no logic. It just maps `WidgetKinds` from the module `UI` to `Widgets` and `ConfItems` of the module `GUI`. The following table sums up this mapping by the function `widgetUI2GUI`:

	UI	**GUI**	
Type:	**WidgetKind**	**Widget**	**ConfItem (special)**
	Col	ColC	
	Row	RowC	
	Matrix wss	MatrixC … wss	
	Label	Label	
	Button	PlainButton	
	Entry	Entry	
	MenuBar, Menu	MenuButton	Menu
	Menu		MMenuButton
	MenuSeparator		MSeparator
	MenuItem		MButtonC
	Canvas w h	Canvas	Width w, Height h
	CheckButton checked	CheckButton	CheckInit init
	ListBox h items sel	ListBoxScroll	Width 10, Height h, List items, CheckInit sel
	CommonListBox	ListBox	
	ScrollBarH uiref	ScrollH guiref	
	ScrollBarV uiref	ScrollV guiref	
	Scale min max	Scale min max	
	TextEdit rows cols	TextEdit	Width cols, Height rows
	NULL	NULL	

A `MenuBar`, resp. a `MenuButton`, is a special case here, as the first `Menu` in a `MenuBar` is mapped to a `MenuButton`'s `Menu`, which is, in contrast to the module `UI`, a `ConfItem` and not a `Widget`. The same applies to the former `Menu`'s children. Other children of the `MenuBar` are ignored.

The extensions for the `NULL`, `CommonListBox`, `ScrollBarH` and `ScrollBarV` *widgets* are trivial and this is also the case for the new properties; thus, we can skip the details.

The next goal is to implement insert, delete, move and configure in the context of the module `UI`.

We would also like to increase the usability of these operations. One aspect to achieve the latter is to get rid of those IO-references, which have to be managed by the user's application (see above).

In order to prepare this, we introduce the type

```
type UIState = IORef UIWidget
```

where the current *UI*-definition can be stored. The module `UI2GUI` already defines a data type

```
data State = State (GUI.GuiPort,[GUI.ReconfigureItem])
```

which is part of an environment:

```
data UIEnv = UIEnv (IORef State)
```

But the name "State" may be misleading, as it suggests a global state, but in fact each event handler creates its own version of it, as the mapping of a *UI*'s command to a *GUI*'s shows:

```
ui2guicmd cmd gp = do
    stateref <- newIORef (State (gp,[]))
    cmd (UIEnv stateref)
    State (_,reconfigs) <- readIORef stateref
    return reconfigs
```

The `State` is used to collect the `GUI.ReconfigureItems`, produced by a *UI*-command. After that, these are returned, as expected from an event handler in the module `GUI`.

Nevertheless, we can extend the `State` with the `UIState`

```
data State = State (GUI.GuiPort,[GUI.ReconfigureItem],UIState)
```

and create the latter in `runUI`, pass it to `widgetUI2GUI` and then to an extended version of `ui2guicmd`. Thus, it is in fact a global state now, as handlers share one reference to the `UIState`. It can now be accessed by any event handler, as the environment is automatically passed to them.

Note that we did not change existing functions, like `setHandler` or `changeStyles`, to update the global state, due to the downward compatibility requirement. We define our own versions of these below.

5.5 GUI2UI

In 5.1.5 functions to create insert, delete, move and configure commands have been implemented. These are based on the *command pattern* and additionally return an updated version of the concerned *widget*'s parent (or the *widget* itself, in case of `createConfigure`). We want to keep these concepts in the *UI*-versions of the operations. Also, it is common sense that the same code should not be implemented more than once, as for example changes had to be applied to any version, which is error prone. Therefore, we reuse the functions. To do so, we have to be able to map elements from the module `GUI` to `UI`.

We implement a new module `GUI2UI`. The mapping is basically the opposite of what is done in `UI2GUI`; thus, we do not show the details here. The table in 5.4 may serve as an overview again.

We initially had another use in mind: translate a program written in the context of the module `GUI` to the module `UI` and then, for example, a *WUI* (`UI2HTML`). But as `UI` is abstract, the concrete types for, e.g., environments and event handlers are not compatible in that direction. But `GUI2UI` is still useful, besides its main purpose, to test a program based on `GUI` in a `UI`-context. Furthermore, a module `GUI2HTML` already exists.

5.6 DynUI2GUI

At this point, everything is ready to implement the operations to dynamically add, remove or configure *widgets* in the context of the module `UI`. We define a new module `DynUI2GUI` for these. We start off with a new type synonym:

```
type ReconfigureItem = (UIWidget, GUI.ReconfigureItem)
```

Referring to 5.1.5, the functions defined in that chapter return a `GUI.Widget` and a `GUI.ReconfigureItem`. The idea is to keep the latter, but translate the former to a `UI.Widget` and update the `UIState` with the result. Thus, we require an operation to process such a `ReconfigureItem`. That's what the following function does:

```
reconfigure :: [ReconfigureItem] -> UIEnv -> IO ()

1  reconfigure items (UIEnv state)
2      | null items    = done
3      | otherwise     = do
4          State (gp, reconfigs, ui) <- readIORef state
5          uiwidget <- readIORef ui
6          writeIORef ui (newui uiwidget items)
7          writeIORef state (State (gp, reconfigs ++ (map snd items), ui))
8              where   newui uiwidget' []           = uiwidget'
9                      newui uiwidget' (item : is) = newui
10                         (updateUI (fst item) uiwidget') is
```

It actually processes a list of `ReconfigureItems`, as one may collect them and execute them at once, but it is also possible to call it more than once. We retrieve the current *UI*-definition in the lines 4 and 5. After that, we repeatedly update it with the first part of each item, in the order they have been passed (lines 8-10) and write it back (line 6). We add the second parts to the list of `GUI.ReconfigureItems` and write back the updated state (line 7). Due to this implementation, we keep track of the *UI*'s state, but hide it to the user. Hence, a user does not have to manage IO-references with the *UI*-definition anymore.

Another aspect to increase the usability is to reduce the amount of information required by an operation. In the following we achieve this by using the functions `getWidgetByRef` and `getParent`, previously defined in the module `UI`. The `ReconfigureItems` are created by the functions `createInsert`, `createDelete`, `createMove`, `createLabelConfigure`, `createHandlerConfigure`, `createStyleConfigure`, `createMenuInsert`, `createMenuDelete` and `createMenuReplace`. We examine `createInsert`:

```
createInsert :: UIRef -> UIWidget -> Maybe Int -> Maybe Int ->
   Maybe Bool -> UIEnv -> IO ReconfigureItem

createInsert parentref child@(Widget kind _ _ _ _ _) mbrow mbcol mbnewrow
    env@(UIEnv state) = case kind of
    Menu            -> case mbrow of
        Just row    -> createMenuInsert parentref child row env
        Nothing     -> error "DynUI2GUI.createInsert: Missing row index!"
    ...
    _               -> do
        State (gp, _, ui) <- readIORef state
        uiwidget <- readIORef ui
        createInsert' gp ui uiwidget parentref child mbrow mbcol mbnewrow
```

The new *widget* is still required to be assigned, but instead of the whole parent, its reference is sufficient here. As menus and menu items are *widgets* in the module `UI`, it seems reasonable for a user to insert them by this function. But in the module `GUI` these are `ConfItems` and thus subjects to `createConfigure`. Therefore, we define different operations for these items (see below), but call them from here (lines 3-6). As we will have to look up the parent in the *UI*'s state, we have to read that here at first (lines 8 & 9). The actual insertion takes place in `createInsert'`:

```
createInsert' :: GUI.GuiPort -> IORef UIWidget -> UIWidget -> UIRef ->
    UIWidget -> Maybe Int -> Maybe Int -> Maybe Bool -> IO ReconfigureItem
```

```
1 createInsert' gp ui uiwidget parentref child mbrow mbcol mbnewrow =
2   return (widget, r)
3     where   parent          = case getWidgetByRef parentref uiwidget of
4                   Just p  -> p
5                   Nothing -> error ("DynUI2GUI.createInsert: Parent "
6                       ++ "not found in environment!")
7             (guiwidget, r)  = GUI.createInsert gp
8                   (widgetUI2GUI ui parent) (widgetUI2GUI ui child)
9                   mbrow mbcol mbnewrow
10            widget          = GUI2UI.widgetGUI2UI guiwidget
```

In line 3 the given parent is looked up (as the new *widget* is not in the environment yet, we cannot use `getParent`). In the lines 7-9 we translate the *widgets* to the context of the module `GUI`, before we use the `createInsert` function from that module. We can then map the resulting parent back in line 10 by using `widgetGUI2UI` defined in `GUI2UI` and return this, together with the `GUI.ReconfigureItem`, in line 2. We perform a quite similar procedure to process move-, delete- or (re)configure-operations.

Inserting a menu or a menu item into a menu bar is a bit different, as it corresponds to a reconfiguration of the latter. Note that it is impossible to directly refer to a `Menu`, `MenuSeparator` or `MenuItem`, because these, as stated above, are `ConfItems` in the underlying module `GUI`; thus, they do not have references and cannot be configured or inserted. This is the reason why the menu bar's reference is always required for the function

```
createMenuInsert :: UIRef -> UIWidget -> Int -> UIEnv ->
    IO ReconfigureItem
```

One could navigate to the desired position in the menu bar, resp. its sub-menus, by defining an index list. But this seems unreasonable, as such a list can become quite complex. Also, a complex menu structure with many nested sub-menus is seldom employed in an application, as it is difficult to navigate in it for the user, too. Hence, we restrict indexing to the top menu only (third argument). We retrieve the parental menu bar (if existent) and the menu (must also exist) with the `UIRef`. Consequently, a menu bar must always be created together with a (single) menu and in order to delete that menu, the menu bar must also be deleted. We are then recreating the menu's content, with respect to the new child and its desired position. Finally, we replace the menu bar's menu with the resulting one. Note that it is still possible to create a complex menu structure, but it has to be defined by oneself before inserting. Alternatively, one can right use `createMenuReplace`, in order to replace the whole top-level menu.

With the basic operations in place, we are now also able to combine them to more complex ones. The function

```
createReplace :: UIRef -> UIWidget -> UIEnv -> IO [ReconfigureItem]
```

considerably profits from the reduction of input information. It just takes the reference of the *widget* to replace and the new one (and of course the environment):

```
1  createReplace oldref new env@(UIEnv state) = case getKind new of
2      Menu    -> do
3          rep <- createMenuReplace oldref new env
4          return [rep]
5      _       -> do
6          State (gp, _, ui) <- readIORef state
7          uiwidget <- readIORef ui
8          createReplace' gp ui uiwidget
9              where createReplace' gp ui uiwidget = do
10                     del <- createDelete oldref env
11                     ins <- createInsert' gp ui
12                         (updateUI (fst del) uiwidget)
13                         parentref new mbr mbc mbnewrow
14                     return [del, ins]
15                         where   parent      =
16                                   case getParent oldref uiwidget of
17                                     Just p  -> p
18                                     Nothing ->
19                                         error ("DynUI2GUI.createReplace: "
20                                         ++ "Parent not found in “
21                                         ++ “environment!")
22                                 parentref   = case getMaybeRef parent of
23                                     Just r  -> r
24                                     Nothing ->
25                                         error ("DynUI2GUI.createReplace: "
26                                         ++ "The widget's parent does not "
27                                         ++ "have a reference!")
28                                 (mbr, mbc)  = findIndexUI oldref parent
29                                 mbnewrow    = case getKind parent of
30                                     Matrix _    -> Just False
31                                     _           -> Nothing
```

We can retrieve the parent's reference (lines 15-27) and even compute the position by using the function `findIndexUI`, we defined in the module `UI` (line 28). The result is then a delete, followed by an insert operation (lines 10-13). Note that we have to use `createInsert'` here, because we have to pass an updated version of the *UI*'s state, as a result of the deletion, in. This would also be possible by calling `reconfigure` in between, but that would already perform one part of the command and thus would pass over the user's decision when or whether that should be done.

We are now revisiting the example from the end of 5.1.5. Reimplemented in the context of the module `UI`, we get the following listing:

```
import DynUI2GUI

widget = row [
        Widget Row Nothing (Just r) [] [] [createButton r 1]
    ]
    where r free

createButton parent n = Widget Button (Just "Button") (Just b) [
        Handler DefaultEvent (Cmd (leftHandler parent n)),
        Handler MouseButton3 (Cmd (rightHandler b))
    ] [] []
    where b free
```

```
leftHandler parent n env = do
    ins <- createInsert parent (createButton parent (n+1)) Nothing
        (Just (n+1)) Nothing env
    reconfigure [ins] env

rightHandler b env = do
    del <- createDelete b env
    reconfigure [del] env

main = runUI "UI Insert Delete Test" widget
```

Compared to the `GUI`-version, we use a reference to the parent, instead of the concrete definition. Due to the implementation of the module `GUI`, the main *widget* cannot have a reference; therefore, we put the `Row` *widget* into another one here. We reduced the number of lines of code from 38 to 23 and it should be quite obvious that complexity has decreased and thus usability has strongly increased.

Finally, to increase the usability even more, we define abbreviations, namely `insertWidget`, `insertMenu`, `deleteWidget`, `deleteMenu`, `moveWidget`, `configureLabel`, `configureHandler`, `configureStyle`, `replaceWidget` and `replaceMenu`. These just call the corresponding createX function and immediately `reconfigure`. They can be used if the command object is not required.

6 Design and Implementation

As we defined the requirements and implemented the basic *UI*-features, we are now designing and implementing the *UI-builder*, which is the actual goal of this work. This involves several steps: designing the architecture and then the basic features, i.e., the core features, as well as the advanced features and implementing these. After that (although the order is irrelevant, as we will see in the next section), as a *UI-builder*'s purpose basically is to generate code from a visual design, this is the final step.

We name the application *FLUID*, which is an acronym for *Functional Logic User Interface Designer*.

6.1 Architecture

In order to establish a well structured and organized system, we define the system's architecture, before we start designing its components. Referring to [Sommerville 2007], "large systems are always decomposed into sub-systems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design. The output of this design process is a description of the software architecture." (p. 242). Consequently, we are identifying the sub-systems now.

Two components can easily be identified:

1. The interface component is the part of the system where the user designs the *UI*. The result is a definition of that *UI*.
2. The parser is responsible for the translation of the *UI*-definition to a standalone *Curry*-program (and maybe back).

But we would like to introduce another one:

3. As an intermediate format, *XML*-files store a *UI*-definition.

This approach has some advantages:

- An unfinished design can be saved and resumed at a later date.
- The interface component is independent from the parser and vice versa.
- Due to the independence, interface and parser are exchangeable. An *XML*-file could be defined by an interface written in a different language, as well as the parser. Furthermore, if, for example, the module `UI` changes, this does not necessarily lead to modifications of these sub-systems.
- An *XML*-file could be developed without any knowledge about the language *Curry*.
- An *XML*-file could define a custom *widget*, which can be imported to a design.

These sub-systems can be seen as layers. Thus, *FLUID*'s architecture naturally results in a *layered model*. Referring to [Sommerville 2007] again, "the layered model of an architecture […] organizes a system into layers, each of which provide a set of services. […] An example of a layered model is the OSI reference model of network protocols." (p. 250f).

Furthermore, besides the advantages mentioned above, "the layered approach supports the incremental development of systems. […] So long as its interface is unchanged, a layer can be replaced by another, equivalent layer." ([Sommerville 2007], p. 251). Due to these, we are able to develop the layers separately from each other and each layer can be useful without the others.

We schematically illustrate the architecture in Illustration 27:

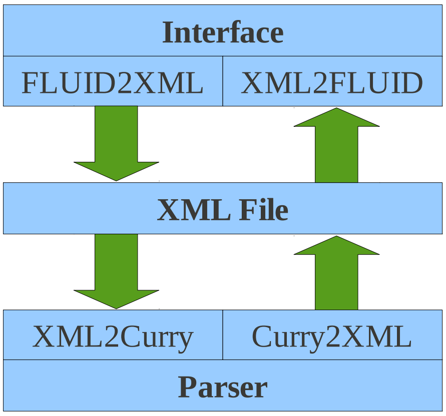

Illustration 27: The GUI-builder's layered model

The interface produces a *UI*-definition and contains an additional parser component. One half stores that definition in an *XML*-file. The other half reads an *XML*-file to resume the design on a definition. Furthermore, an *XML*-file could define a custom *widget*, which can be imported by the interface into a design. The parser generates *Curry*-source code from an *XML*-file and writes that into another (*.curry-)file. Contrary, it creates an *XML*-file from a *Curry*-program.

6.2 Overview

The next step is to define the basic application. We identified the most important components in chapter 3. We can also derive a *GUI-builder*'s general structure and thus *FLUID*'s initial layout:

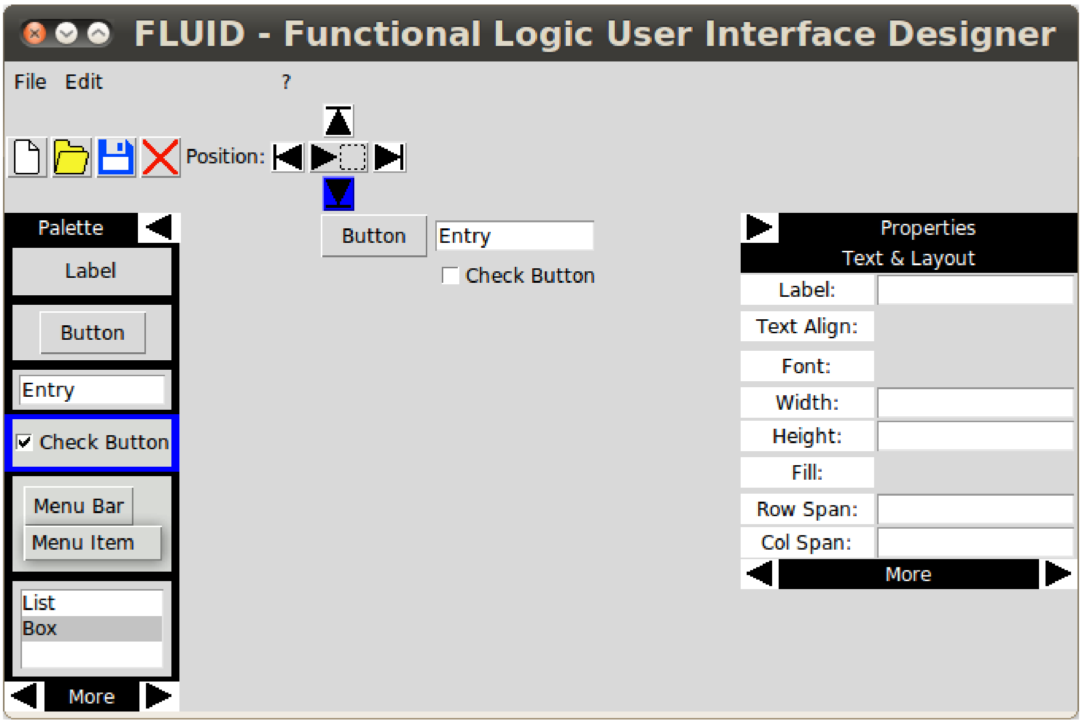

Illustration 28: FLUID - overview

At the top is the *Menu Bar* and the *Tool Bar* below. On the left is the *Palette*, the *Design View* is in the center and the *Properties* on the right. In the following, we give a rough description of these components and go into detail in 6.3.

In the menu *File* a new design can be created, a previously saved design opened, the current design saved, source code generated from any design or the application can be quit. In the menu *Edit*, the mode can be switched from *Insert* to *Move* and vice versa, the insert or move position be chosen and a *widget* can be deleted. Furthermore, a help and the about can be accessed in the menu *?*.

The *Tool Bar* is quite important, because, referring to 4.2, we cannot use context-menus to provide some options, as in NetBeans or Visual Studio and thus the *Tool Bar* is an alternative. But any option in the *Tool Bar* (so far *New*, *Open*, *Save*, *Delete*, *Position* and the mode) is also available in the *Menu Bar*. Both should be self-explanatory.

A *widget* to insert can be chosen in the *Palette*. There are various *widgets* and they are represented by images in order to provide a preview.

The *Design View* displays the *UI* to create and provides access to the overall layout, i.e., the arrangement of *widgets*.

A *widget*'s configuration can be customized in the *Properties*. They cover general properties, like the reference and event handlers, text and layout properties, like a *widget*'s label or the column span, properties regarding the appearance, like the background or border, as well as special properties for a `MenuBar` or a `ListBox`.

6.3 Core Features

We are now implementing the components roughly described above. We define a new module `FLUID` for the main application. In order to keep track of *FLUID*'s current state, we define type synonyms of IO-references, which can store the relevant information and be accessed by any event handler (if passed in). We require a state for any component:

- `type PaletteState = IORef (Maybe (UIRef, UIRef -> UIRef -> UIWidget))`

  The `PaletteState` observes the currently selected *widget* kind in the *Palette* and a function to create that *widget* from a reference. If *FLUID* has just been started, nothing is selected, i.e., the state contains `Nothing`.

- `type ToolBarState = IORef (InsertPosition, Mode)`

  The `ToolBarState` stores the insert or move position and whether the mode is *Insert* or *Move*.

- `type DesignState = IORef (UIRef, [(UIRef, [Style])])`

  We keep track of the root or main *widget*'s reference, as well as a list of selected *widgets* in the design, together with some of their styles (see below), with the `DesignState`.

- `type PropertiesState = IORef [(UIRef, [Style])]`

  As *FLUID* shall be a desktop application, it is based on the *GUI*-part of the *UI*-library, which lacks an implementation for some styles and just ignores these. Hence, some styles defined in the module `UI` are not applicable to our *live widgets*, resp. would get lost. Therefore, the `PropertiesState` saves a list of these styles, which can be applied later.

- `type ReferencesState = IORef [(UIRef, String)]`

  We store a mapping from a *widget*'s reference to a reference name in the `ReferencesState`. These names are required for event handlers, which are, in most cases, changing a *widget*'s state and are not directly applied to the *live widget* when the user defines them, but when the source code is generated and thus when the actual references are not available anymore. An event handler then takes the name as an argument. The creation of the reference names is very simple: We just delete the prefix of the reference, which is referring to the root *widget*'s parents, as these won't exist in the final *UI*. We update this state whenever a *widget* has been created, deleted or renamed.

- `type ArgumentsState = IORef [(String, String)]`

  External arguments are sometimes passed to a (main) *widget*, e.g., a database handle, which have to be accessed by event handlers. The user can define these arguments for the main *widget* and they are saved in the `ArgumentsState`. The type of the argument, together with the defining module, must also be assigned, e.g., "DB.Database". The latter is especially relevant for source code generation, but may also help the user to keep track of the arguments.

- ```
  type HandlersState =
      IORef [(UIRef, [(String, String, [(String, String)])])]
  ```

  Furthermore, the `HandlersState` contains a mapping of event handlers to *widgets*. The user can define a handler to call when an event on the *widget* fires. This definition covers the type of the event, the command, i.e., a function's name, as well as a list of the handler's arguments. The arguments again have a type.

- ```
  type FLUIDState = (PaletteState, ToolBarState, DesignState,
      PropertiesState, ReferencesState, ArgumentsState, HandlersState)
  ```

  Finally, the `FLUIDState` combines all of the above to a tuple; thus, they are easier to handle and types become shorter.

6.3.1 Palette

The *Palette* is a simple list of *widgets*, which can be inserted into a design. As we would like to provide a preview of the *widgets*, a list box or another predefined *widget* is not sufficient. We put a list of `Labels` with images showing the corresponding *widgets* into a column instead. Due to the missing border property for *GUI-widgets*, we set their size a few pixels wider and taller than the image's and assign a black background, which substitutes the border. When the user selects one of the `Labels`, its background color is changed to blue. In order to restore its background later, its reference, as well as the corresponding create-function are stored in the `PaletteState`. Such a create-function, e.g.,

```
createEntry state prefs ref _ = Widget Entry
    (Just "Entry") (Just ref)
    (createDefaultHandlers ref state prefs)
    [Class [Width 12, Bg White]] []
```

also applies some default properties, like `Width` and `Bg` and event handlers (see 6.3.2) to the *widget*. Furthermore, the function takes the `FLUIDState` and a list of references, called `prefs`, for some of the *widgets* in the *Properties*. Both are relevant to event handling. Finally, it takes a reference for the *widget* to create and another one, which is solely used by scroll bars, as these require a target.

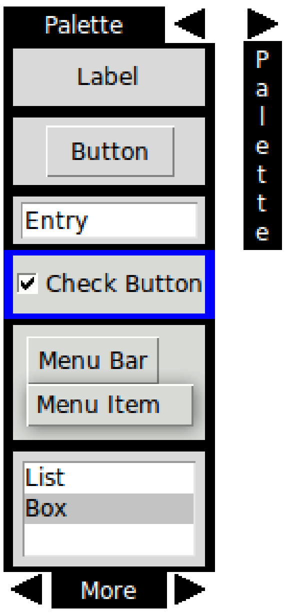

Illustration 29: Palette: shown and hidden

What generally complicates *FLUID*'s design is that, referring to 4.2, screen space is limited and cannot be expanded by scroll bars in most cases. As a result and because a user's design may take most of the available space, a new (major) requirement arises for the components described above: In the direction of the *Design View*, they have to be as narrow or flat as possible. This is especially the case for the *Palette* and the *Properties* as these project into the *Design View* and limit the space for the design. We also have to find a solution for a long list of *widgets*, where some of them may not fit into the available space, especially at low screen resolutions.

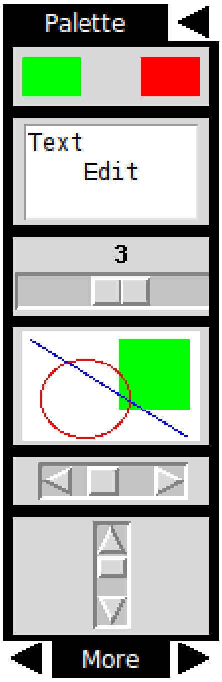

Illustration 30: Palette: second page

In order to save space for the *Design View*, we can, of course, design the *Palette* and *Properties* narrow, to some degree. Another useful feature for this purpose is the possibility to hide these components. When the button (actually, it is a `Label`, as it fits better) next to the label “Palette” is pushed, the *Palette*'s `Display` property is set to `False`. A narrow `Label` is shown instead, see Illustration 29.

We solve the problem with a tall component running out of the screen in a similar way. For the *Palette*, for example, we organize the labels in different columns or pages and these into a row and set just one of the pages' `Display` properties to `True`. With a click on one of the buttons next to the label “More”, one can turn over the page, i.e., set the current page's `Display` property to `False` and another one to `True`. We put the more important *widgets* on the first page. Illustration 30 shows the second. Due to these features, the *Palette* should fit to the usual screen resolutions.

6.3.2 Design View

The *Design View* (Illustration 31) is the component which contains the design. Besides the design (1), there also are two empty `Rows`, one to the right of the design (2) and one below (3). Their purpose is to initialize the *Design View*'s size and to take any excess space, which is not required by the other components. Otherwise, the design would stretch until it takes all the available space, disregarding the size properties of the *widgets* it contains. We initialize the design itself with a label in a `Matrix`. The former will be replaced by the first *widget* the user inserts, i.e., it is a placeholder. Thus, the initial *Design View* looks as follows (the red lines are for visualization only):

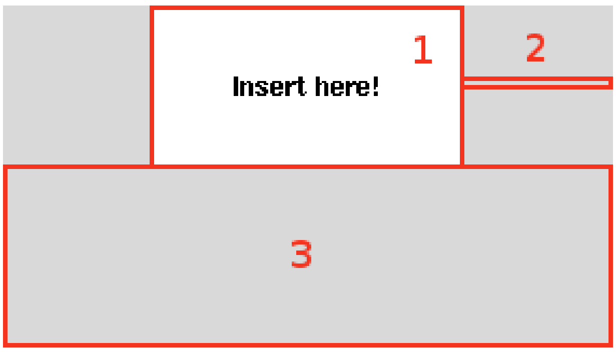

Illustration 31: The initial Design View

(2) has a height of zero and the `Fill X` property and therefore is responsible for the expansion in the x-direction only. If it had `Fill Both`, (1) and (2) could stretch down to the bottom of the main window. For (3), on the other hand, we set an initial width and apply the `Fill Both` property.

So far, we limit the design's layout to just a single `Matrix`, as it is powerful enough to achieve most designs one can imagine and simplifies the implementation, especially regarding index computations. With nested layouts, we also had to implement a mechanism to select a *widget*'s parent. We would like to delay this until we introduce custom *widgets* later.

There are three features the design must provide:

1. Inserting a *widget*.
2. Selecting a *widget*.
3. Moving a *widget*.

Inserting a Widget

The first one is probably also the most difficult. The first step is to find some graphical representation for the layout, which the user requires to control the design. Like the GUI Builder, which is related to *FLUID*, as both are based on *Tk*, we could implement a grid-based visualization strategy. Actually, we use a grid without visualizing it, but take a more direct approach. As we are employing *live widgets*, we can assign event handlers to the *widgets* in the design (`createDefaultHandlers` in 6.3.1). Therefore, we abandon the visualization of cells as proxies, but use the concrete *widgets* to relatively insert or move another one. I.e., on a right-click on a *widget*, with respect to the chosen mode, a new *widget*, corresponding to the choice in the *Palette*, is inserted to the left, right, top, or bottom of the one clicked, Depending on the position, which has been chosen in the *Tool Bar* or the *Menu Bar*. The result is a mix of the grid-based layout in the GUI Builder, the *GridBagLayout* in NetBeans and some aspects of the *Free Design* in NetBeans or the standard layout in Visual Studio.

An advantage of this approach, compared to the grid-based, is that neither we, nor the users have to manage cells. A *widget* can just be inserted or moved, without creating new cells, rows or columns before; thus, the result is basically the same, but can be achieved in a more direct way.

In a usual grid-based approach one can leave a cell, row or column empty, in order to create empty spaces between *widgets*. We can achieve the same by inserting an empty `Row` (a `Col` or a `Matrix` would produce the same result, as all of them are translated to a frame in *Tk*) and assign the background color; hence, it becomes "invisible". Therefore, we add such a *widget*, we call *frame* in the following, to the *Palette.* It can then be inserted, moved and configured as any other *widget*. See below for an example.

Selecting a Widget

A left-click on a *widget* in the design selects it. We store the references of selected *widgets* in a list in the `DesignState`, where they can be accessed by other components, especially the *Properties*. As it is very easy to achieve, it is already possible to select multiple *widgets*. Because there is no way to define events like the usual <control + left mouse button> yet, multiple *widgets* can just be selected by left mouse clicks.

Selecting a *widget* should also highlight it. As borders have not been implemented for a *GUI* in the *UI*-library yet, which would be the usual way to highlight, we apply background and foreground colors instead. We set the background to blue, the foreground to white and disable an image, which may be connected to the *widget*, because it would override the other changes. When the *widgets* are deselected by left-clicking one of them again, we have to restore the original properties. Therefore, we also store the background, foreground and image in the `DesignState` when a *widget* is selected.

In order to configure the whole design, which corresponds to the main *widget*, it must be selectable, too. As other *widgets* may cover the main *widget* and it would then not be accessible, it is not directly selectable at all. The main *widget* is selected when no other *widget* is, instead. We also assign an event handler, which deselects any *widget* and thus selects the main *widget*, to the empty space below the design ((3) in Illustration 31).

Moving a Widget

A selected *widget* can also be moved, if the move-mode has been activated in the *Tool Bar* or the *Menu Bar*. A move is basically a combination of cut and paste and thus an advanced feature, but we already add it here, as we think it is quite important in order to accurately arrange *widgets*. The process is nearly the same as inserting a *widget*, just the computation of the *widget*'s new position is a little bit different, due to the implementation of `moveWidget` in `DynUI2GUI`. So far, only the *widget* which has been selected at last can be moved.

6.3.3 Properties

The *Properties* is the the most complex, but also the most important component. The configuration of selected *widgets* (so far, just the one selected at last) can be adjusted here. As there are many options, we also implement the page-system, we already used for the *Palette*. The *Properties* can be hidden, too.

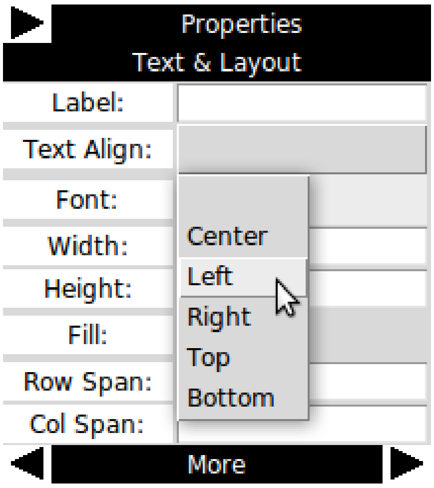

Illustration 32: Basic properties

From a developer's perspective, we can put the properties into four different categories:

1. Basic properties, like a widget's label, `Height`, `Fill` and so on, which can be directly applied.
2. Basic properties, which cannot be directly applied, either because we have used them to highlight a selected *widget* and thus they would disturb the selection mechanism, these are `Bg` (background), `Fg` (foreground) and `Image` or they would break the selection mechanism, these are `Active` and `Display`, or they have not been implemented for a *GUI*, but in the module `UI` and would get lost due to the implementation of `UI2GUI`, these are `Font` and `Border`.
3. Properties for a special kind of *widget*.
4. General properties regarding event handlers and code generation.

Basic Properties

We implement properties in the first category as simple key-value pairs, i.e., a `Label` and an `Entry` or a drop-down menu, see Illustration 32. The user can type the value into the `Entry` and assign it to the selected *widget* by pressing the return-key or choose the value in the menu. This will call `DynUI2GUI.configureStyle` and immediately apply the change. An empty value removes the property. The drop-down menu is basically a `MenuBar`, but with a special configuration. As we employ it quite often, we implement a function

```
createStringChoice :: UIRef -> [String] -> Int -> [Style] ->
     (String -> UIEnv -> IO ()) -> UIWidget
```

which creates such a menu with the given reference from a list of values. The `MenuBar`'s initial value is the value at the given position in the list. There can also be styles, like `Height` or `Width`, applied to it. When the user selects an option, the given event handler is called with the corresponding string as an argument.

The only problematic properties in this category are `RowSpan` and `ColSpan`. As a spanning *widget* shall not overlap another one, we have to insert `NULL` *widgets*. We compute the difference of the old and the new value, in order to determine the amount of *widgets* to insert or delete. If there is either the difference for `RowSpan` or `ColSpan` greater than one or less than zero, we can just fill up the cells below or to the right of the spanning *widget*, resp. delete `NULL` *widgets* there. Complexity increases if this predicate applies to both differences. In this case not just a row or column, but an area has to be filled up, resp. deleted. Because of this relationship between the two properties, we always process them together and they can also be assigned by the user at once. Computing the required position indices is a little tricky, but shall not be examined in detail here.

A *widget*'s `RowSpan` and `ColSpan` are also relevant for the insertion of a new *widget* on the former or for the movement of an existing *widget* on the spanning, as well as moving the spanning *widget* itself. We have to treat a spanning *widget* as a combination of its `NULL` *widgets* and the *widget* itself; thus, we have to add `RowSpan` and `ColSpan` values to the insert or move position in order to move or insert to the right or below the spanning *widget* and its `NULL` *widgets*. Furthermore, we have to move a spanning *widget* together with its `NULL` *widgets*. As this is quite complex, we would like to find an easier solution. In fact, there is a simple one: We temporarily set the *widget*'s `RowSpan` and `ColSpan` to one and thus also delete the `NULL` *widgets*, until the changes have been applied and then restore the values, to reinsert the latter.

Inapplicable Properties

Similar to the first category, we represent options from the second one in the *GUI*, but the results differ. As a selected *widget*'s background, foreground and image have been saved in the `DesignState`, we can directly manipulate these values. The changes will take effect when the *widget* is deselected and the values restored. Analogue to `createStringChoice`, we implement

```
createColorChoice :: UIRef -> [Color] -> Int -> [Style] ->
    (Color -> UIEnv -> IO ()) -> UIWidget
```

which provides a drop-down menu to choose colors for the background and foreground. The other properties in this category cannot be directly applied, but we can store them in the `PropertiesState` and delay their application, until the design is saved or source code generated.

Special Properties

So far, there are two kinds of *widgets* requiring special properties, namely the `CommonListBox` and the `MenuBar`. Their content can be configured on a separate page in the *Properties*. For the `CommonListBox` this is its items, i.e., the `LBList`, as these aren't *widgets* and thus cannot be configured like one. Illustration 33 (left) shows the list box' properties:

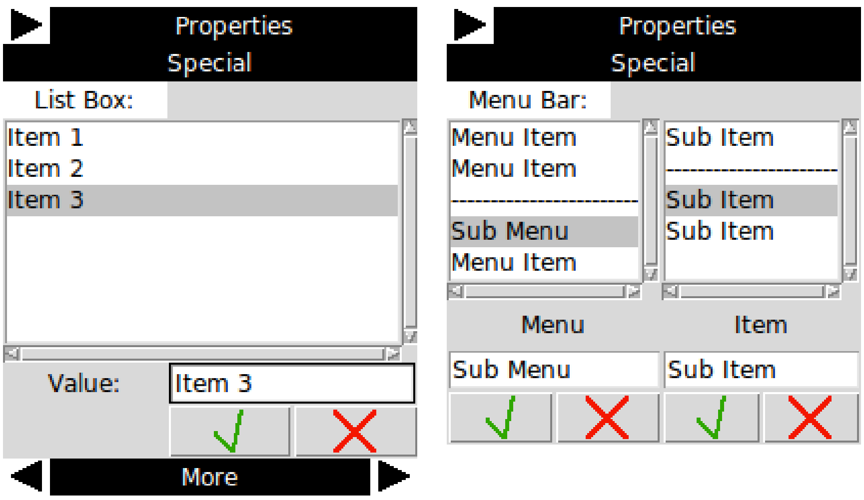

Illustration 33: Special properties for a CommonListBox (left) and a MenuBar (right)

An item can be added behind the selected one by typing the value in and pressing the return-key. A selected item can be updated by pressing the tick-button or deleted by pressing the cross-button. The system immediately applies any change to the *widget*. We can keep the current values over different events by reconfiguring the event handlers with these values as an argument, whenever a change has been applied. Due to this approach we realize a kind of a side effect.

Configuring a `MenuBar`, i.e., `MenuItems`, `MenuSeparators` and `Menus`, works very similar. Indeed these are *widgets* in the module `UI`, but we cannot configure them like one, because they do not have references. We connect them to the parental `MenuBar`'s properties instead. An item can be added, updated and deleted. In addition, one of the three types has to be chosen. A `MenuItem` can be configured like a list box' item. A `MenuSeparator` is represented by a dashed line in the *Properties* and its text is ignored. A `Menu` is different, as it may have items itself. In order to keep it simple and as we have already stated above, a complex menu structure is difficult for both, the developer and the user, we limit the configuration to the main menu and a single sub-menu. The sub-menu is displayed next to the main menu, as shown in Illustration 33 and can be configured like the latter. As above, changes are immediately applied.

General Properties

Finally, there are general properties, see Illustration 34. Any *widget* should have a reference, as they are required to control a *widget* by an event handler, for example by a handler, which is called on a submit-button and takes the content of an `Entry`. Hence, one of a *widget*'s most important properties is its reference or more precise the reference name, as it is not identical to the actual reference of the *live widget*. It is generated for any new *widget*, but can also be changed to a more meaningful value. In order to unambiguously identify a referenced *widget*, the reference name must be unique, i.e., another name or argument with the same value shall not exist; therefore, we ensure that before a new name is assigned. The result is stored in the `ReferencesState`.

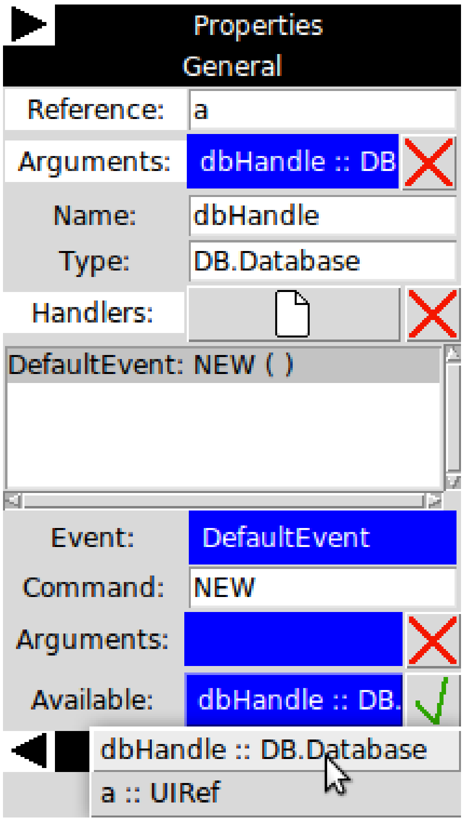

Illustration 34: General properties

As stated above, the main *widget* may also have arguments. These can be defined in the general properties and referenced by user-defined event handlers, as well. Like the reference name, we check if an argument's name is unique and if its type is non-empty. We save arguments in the `ArgumentsState`. If the selected *widget* is not the design's main *widget*, we hide the arguments section.

An event handler or more precise a call of an event handler, as we are only generating this call and not the function itself, consists of an event to fire it on, a command, i.e., the name of the function to call and the arguments of the latter. These values can be defined in the general properties, too. As shown in Illustration 34, there is a pretty printed list of handlers, where one can be selected and its configuration changed below. We assign a default command and event to a new handler, which has been created by clicking the file-button. Reference names, as well as arguments defined by the main *widget*, can be added to the handler's argument-list. Whenever one of these values is changed or deleted, we also update any handler referring to it by searching through the `HandlersState`. A selected event handler can be deleted via the cross-button above the handler-list.

There is a nearly identical view for a `MenuItem`'s handlers on the "Special"-page, if the currently selected widget is a `MenuBar`. As stated above, a `MenuItem` is connected to a `MenuBar`'s properties. Hence, this is also the case for an item's event handlers. Therefore, we just add these to the `MenuBar`'s handlers, but mark them for the item. The only reliable and unique information we have about an item is its position in the menu. Thus, we mark the corresponding handler by adding an index list as a prefix to its command. Illustration 35 shows an example.

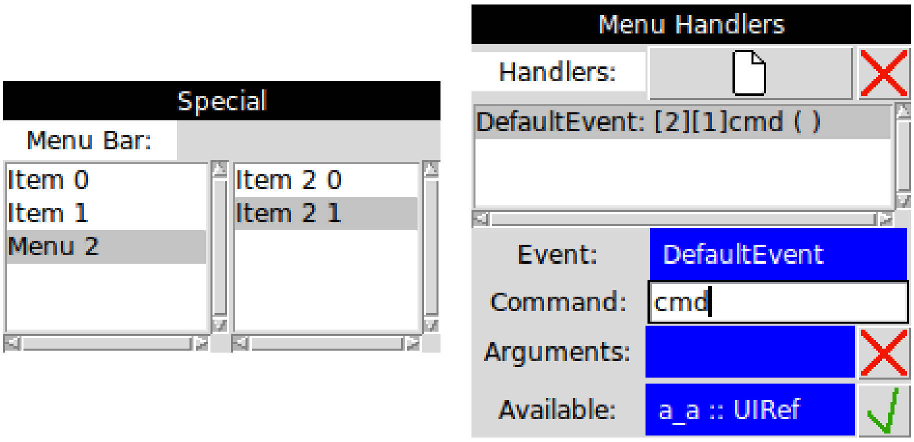

Illustration 35: Configuring a MenuItem's event handler

The second item in the menu at position two in the main menu has been selected, which results in an index "[2][1]". As the illustration indicates, this is automatically assigned to the command; the user does not have to manually add it. The handler can also be defined in the `MenuBar`'s general properties, but this is less comfortable, as the index has to be manually defined. This is also quite unsafe, because the index could refer to an inexistent item and we do not perform any check there.

Mutual dependencies between the general handlers and the menu handlers sections, as a new or updated handler in the one should immediately show up in the other, add some complexity to their implementation.

6.3.4 Example

FLUID has now reached a state where we are able to design the example *GUI* from chapter 3. We do this step by step:

Step 1: At first, we choose a `MenuBar` in the *Palette* and set the mode to *Insert*. A right-click on the empty design inserts the menu.

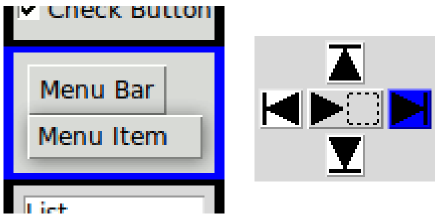

Illustration 36: Example: step 1

Step 2: In the next step, we set the insert position to below, insert a *frame* by selecting it in the *Palette* and right-click the menu to produce the empty space below it. We also insert an `Entry`, another *frame* and the first column of buttons.

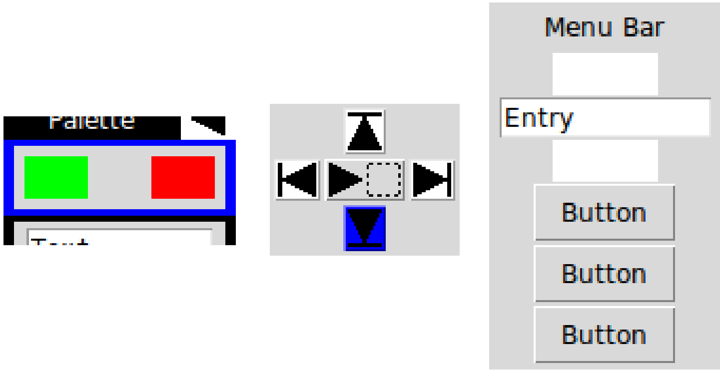

Illustration 37: Example: step 2

Step 3: After switching the position to right, we insert the rest of the buttons and another *frame* behind this column.

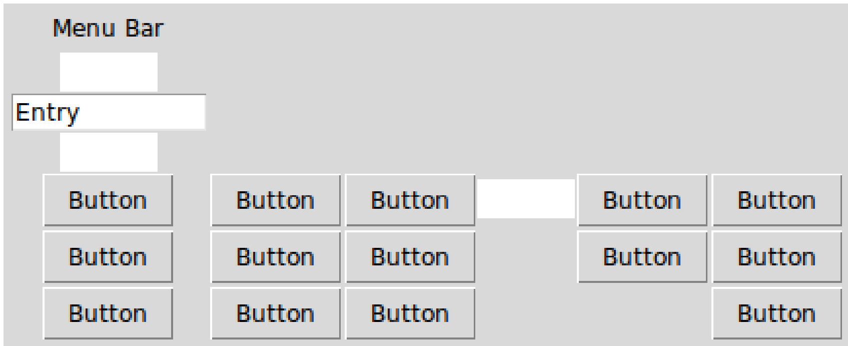

Illustration 38: Example: step 3

Step 4: The next step is to configure any *widget*, here, for example, the plus-button's `Label`, `Fill` and `RowSpan` properties. We also set the *frames*' `Bg` properties to the default color and adjust their sizes.

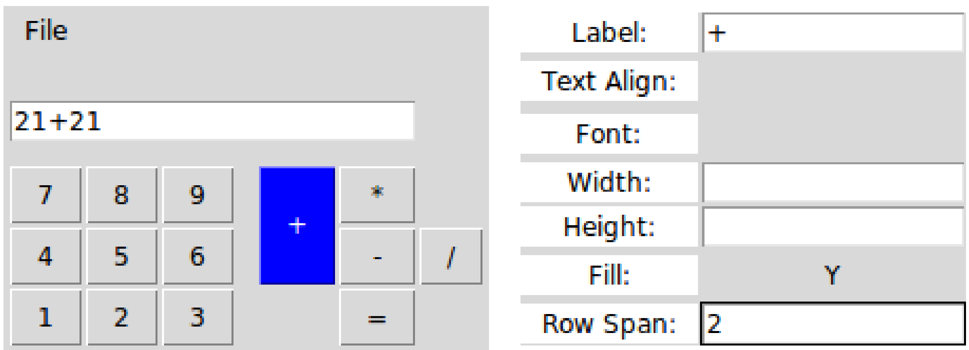

Illustration 39: Example: step 4

Step 5: Because the plus-button spans two rows, a `NULL` *widget* is inserted below it and the minus- and division-button are moved to the right. Therefore, we set the mode to *Move* and move the minus- below the plus-button.

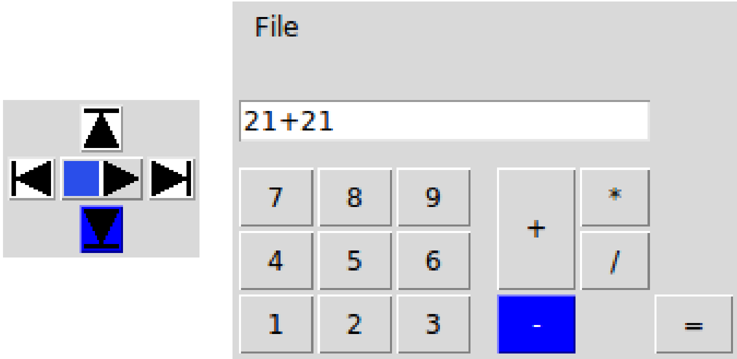

Illustration 40: Example: step 5

Step 6: As there was a `NULL` *widget* below the plus-button, the equals-button is moved to the right, this time. By moving it to the right of the minus-button, we finish the design.

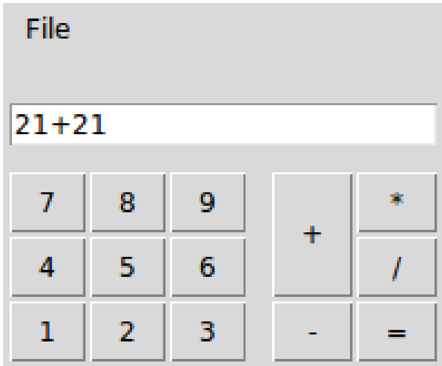

Illustration 41: Example: step 6

We stated before that one advantage of the *UI*-library is its high level of abstraction and that a *UI* can be run as a *GUI* and a *WUI*, without touching the source code (except for the import declaration). While this is basically true, there are some differences regarding their appearance. An important one is the unit of measurement for a *widget*'s size. In a *GUI* this is generally the size of a character, if the *widget* has a string value assigned to it or pixels, if it has not. Hence, a button with just a text is measured in characters, but with an image in pixels. In a *WUI*, on the other hand, a *widget* is always measured in “em”, i.e., characters. Another difference is, of course, that *widgets* may generally differ regarding their appearance, but sometimes also their behavior, because identical *widgets* do not exist. These differences have two impacts on our example, we have to take into account:

1. A `MenuBar` is much wider in a *WUI* than in a *GUI* and its size cannot be directly changed.
2. A *frame* is measured in characters; hence, it is less flexible as a placeholder and its size grows (too) fast with the values of `Width` and `Height`.

We can solve number one by increasing the file-menu's `ColSpan` to four and number two by setting the *frames*' `Width` and `Height` to one. Note that the latter results in one pixel wide and tall *frames* in *FLUID*, what makes them very difficult to select again. Also note that, as `NULL` *widgets* are ignored in `UI2HTML`, we have to add a *frame* behind the six- and the three-button, too. Illustration 42 shows the tweaked design and the resulting *WUI*.

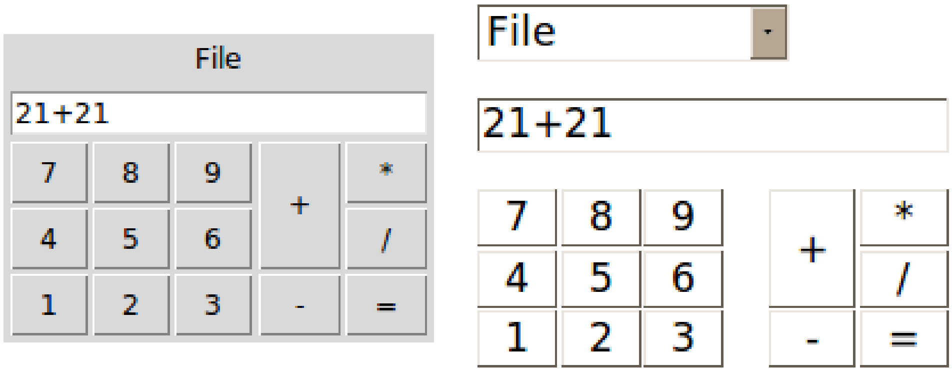

Illustration 42: The example designed for a WUI (left) and the resulting WUI (right)

6.4 XML-Files (Persistent Designs)

As stated above, we would like to save designs in an *XML*-format and be able to load it into *FLUID* at a later date, which is what happens when the load resp. the save option is chosen in the *Menu Bar* or the *Tool Bar*. Before a design is saved, we just have to check whether event handlers are consistent, i.e., whether the arguments of handlers with the same command have the same types. Otherwise the compilation of the source code, we will generate from the *XML*-document later, could fail. This is also the main reason why user-defined arguments require types (references are always of the type `UIRef`).

Nevertheless, the translation into or from *XML* is straightforward with the module `XML`, which is provided by *PAKCS*. We define two new modules: `FLUID2XML` parses a design, together with related states, to an *XML*-document and `XML2FLUID` does the opposite, i.e., restores design and states.

6.4.1 FLUID2XML

The module `XML` defines two constructors for an *XML*-expression of the type `XmlExp`:

```
data XmlExp = XText String | XElem String [(String, String)] [XmlExp]
```

The first one is irrelevant, as we do not use it. The second represents an element or tag with a name, a list of attributes in a key-value format and a list of sub-expressions or children. We would like to imitate the *widget* structure of the module `UI`. Hence, we represent *widgets* by `Widget` tags, which also applies to the document's root element. Other values are either defined as a *widget*'s attributes, if there can just be a single one, like a label or as a child, if there can be multiple, like a `Style`. The initial values for some of the *widget* constructors, especially a `ScrollBarH`'s or a `ScrollBarV`'s target, are mapped to attributes, too. While parsing the *UI*-definition, we can also add the values from the `PropertiesState`, the `ReferencesState`, the `ArgumentsState` and the `HandlersState` to the document. The reference names also serve as targets for the scroll bars. Arguments are solely added to the root element. As a `Matrix` contains a list of lists of children, which cannot be directly mapped to *XML*, we define a tag `Row` for these.

The following example, which is shown in Illustration 43, should clarify the mapping. Note the list box' `refname` (first `Row`) and its usage in the delete-button's handler (second `Row`).

```
<?xml version="1.0" standalone="yes"?>

<Widget kind="matrix" refname="a" label="">
  <Row>
    <Widget kind="common_list_box" refname="listbox" label="">
```

```
      <Style kind="background" color="gold" />
      <Style kind="width" value="12" />
      <Style kind="list_box_list">
        <Item value="Item 1" />
        <Item value="Item 2" />
        <Item value="Item 3" />
        <Item value="Item 4" />
        <Item value="Item 5" />
      </Style>
      <Style kind="height" value="4" />
    </Widget>
    <Widget kind="scroll_bar_vertical" refname="a_b" label=""
        target="listbox">
      <Style kind="fill" direction="y" />
    </Widget>
  </Row>
  <Row>
    <Widget kind="button" refname="deleteButton" label="Delete">
      <Style kind="col_span" value="2" />
      <Style kind="fill" direction="x" />
      <Handler event="DefaultEvent" command="deleteHandler">
        <Argument name="listbox" type="UIRef" />
      </Handler>
    </Widget>
    <Widget kind="null" refname="a_d" label="" />
  </Row>
</Widget>
```

Illustration 43: XML example

6.4.2 XML2FLUID

As stated above, this module just does the opposite of `FLUID2XML`, with one exception: it differs a little bit regarding references. The reason is that there is one case, except for a *widget* itself, where we require the actual reference and not just the reference name. This is a scroll bar's target, which can only be applied when we are creating the scroll bar and it is absolutely possible that the latter has to be created before the former (if it is somewhere to the left or above the target in the tree). Hence, we cannot parse the *XML*-document to the *UI*-definition and the states in one pass, but have to create the `ReferencesState` in the first run and the rest in a second. But we can just map a reference name stored in a `Widget` tag to a free variable and bind the reference to a value later, when the corresponding *widget* is created.

6.5 Optimizations

FLUID is now capable of designing a *GUI* and saving and loading a design respectively. But when the application is run, there is often a well noticeable delay between a command and the corresponding action. For example, if a *widget* is selected in the *Design View*, it needs some time, where the *widget*'s properties are prepared, until the *widget*'s background and foreground switch to the selection colors. Of course, one reason is the amount of properties, which just need time to be set up, but another is the time a *widget*'s lookup requires. We can use `setValue`, `changeStyles` and `setHandler` to reduce the delay where we do not need the *UI*-definition. These operations are faster than the ones defined in `DynUI2GUI`, as they do not perform lookups or updates on the `UIState`. This is, however, not an option for any changes of the *widgets* in the *Design View*, because we require a complete *UI*-definition for these. We would like to increase the speed of `getWidgetByRef` and `getParent` instead, which are frequently used when the design is changed.

A *UI*-definition can be seen as a tree, where each node corresponds to a *widget* and an edge to a parent-child relationship. So far, the lookup operations are based on a naive depth-first-search strategy on this tree. I.e., in order to find a *widget*, we start at the root and follow branches from left to right to their leaves, until the current node's reference equals the *widget*'s we are looking for. Hence, in the worst case, where the target is the rightmost leaf, we have to visit any node and any edge in the tree. This can especially be a problem in *FLUID*, as the *Menu Bar*, the *Tool Bar* and the *Palette* are to the left of the *Design View* and thus are often unnecessarily searched. Illustration 44 shows an example. Nodes are labeled with the corresponding *widgets*' references. Yellow nodes are visited and numbers indicate the visit order.

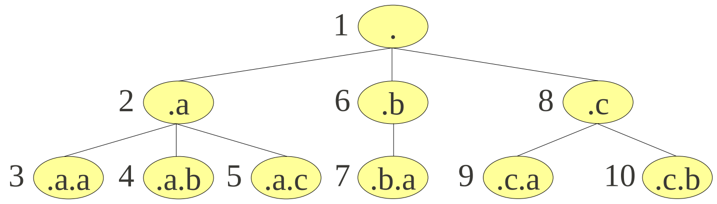

Illustration 44: A naive depth-first-search strategy (searching .c.b)

It is possible to improve the lookup strategy for a *GUI* (module `UI2GUI`): A *widget*'s reference contains hierarchy information regarding its parents, we did not take into account yet. Due to this, we are able to identify whether a *widget* is a sub-*widget* of another and thus if they are in the same branch in the *UI*-tree, by comparing just the references' prefixes. If prefixes differ, an examination of that branch is unnecessary. Otherwise, we check if the suffixes are also equal and thus we have found the *widget* or descend in the branch if they are different. See Illustration 45 for an example.

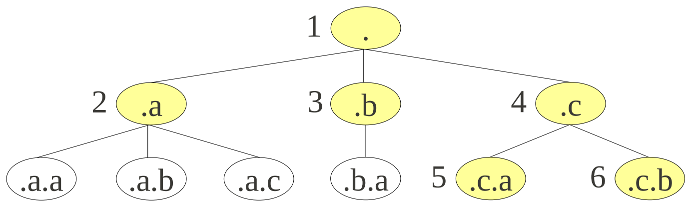

Illustration 45: An improved search strategy (searching .c.b)

Another improvement can be made by reducing the number of widgets. This is especially possible for the *Properties*, as we have put, for example, basic properties like a label, `Height` and `Width` in a single `Row` *widget* each, while we iteratively implemented them. We can replace several `Rows` by a single `Matrix` in most cases and so also simplify the layout a little bit.

Finally, the *Properties* can be directly optimized. The page-system has been explained above. A consequence of this system is that just a part of a *widget*'s properties is visible at a time. Therefore, only the visible part is computed and the other parts not until they are required, i.e., when the user turns the page.

All these optimizations are noticeable, but unfortunately the delay is still there. A reason might be that the underlying *Tcl*-program slows down with this amount of widgets, too. *FLUID*'s quite long startup time and phases where no translations from *Curry* to *Tcl* seem to be processed, are indications for this assumption.

Further improvements could make use of multiple threads and a multi-core CPU.

6.6 Advanced Features

The implementation of *FLUID* now covers the major requirements, as well as some of the minor. In the following section we define the advanced features, which realize more of the minor requirements and some optional ones.

6.6.1 General Changes

Practice and tests have shown that, while the page-system is sufficient for the *Palette*, it may sometimes be useful to directly access more than one page of the *Properties*. Therefore, we replace this system by a similar one, where related properties are distributed to sections instead of pages and each section can separately be shown or hidden. However, if, due to the screen resolution, there is not enough space available for more than one section, other sections can be hidden again. Illustration 46 shows the new system.

As before, just the visible properties are computed and hiding any section increases the performance and further reduces the delay when selecting *widgets*. Hence, this is quite useful when a large amount of *widgets* shall be selected and rearranged.

Furthermore, some icons have been reworked, in order to clarify their meaning.

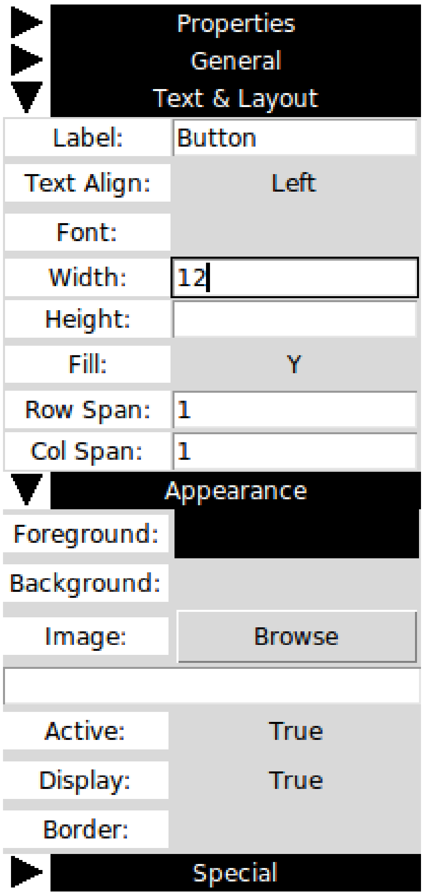

Illustration 46: Properties with sections instead of pages

6.6.2 Moving Multiple Widgets

If a design is based on a single layout, a common use case is that multiple related *widgets* have to be moved. Moving them one by one is unreasonable; thus, it should be possible to move them all together. So far, a selected *widget* is moved relative to a target by choosing the position and right-clicking the latter. We would like to reuse this implementation. Multiple *widgets* can already be selected and stored in the `DesignState`, but the order they are selected in is random and, although they are somehow positioned relative to each other, the `DesignState` does not reflect this relation. Therefore, the first step is to retrieve the selected *widgets*' absolute positions with `findIndexUI` and sort these from top to bottom and left to right and into lists representing rows. After that, the relative positions can be computed, which yield the targets of the move-operations. The upper left *widget* is used as an orientation point; hence, its target is the original target of the move. Successive *widgets* in that row have the preceding one as their target and are moved to the right of the latter. The first *widget* in the following row targets the first one in the row before and is moved below that and so on. If the *widgets* shall be moved above the original target, we just reverse the list of rows and move above instead of below; thus, the lower left *widget* then is the orientation point and rows are moved bottom-up.

Due to the translation from absolute to relative positions, we can also maintain the latter, if there are *widgets* between the selected ones. Illustration 47 shows an example. The buttons “B1”, “B2”, “B3” and “B4” are moved as a compound below the label “Target”. The order they have been selected in is irrelevant. Note that move positions are based on the positions before moving. That is the reason why the buttons finally are not directly below “Target”, because it has been moved to the original position of “B3”.

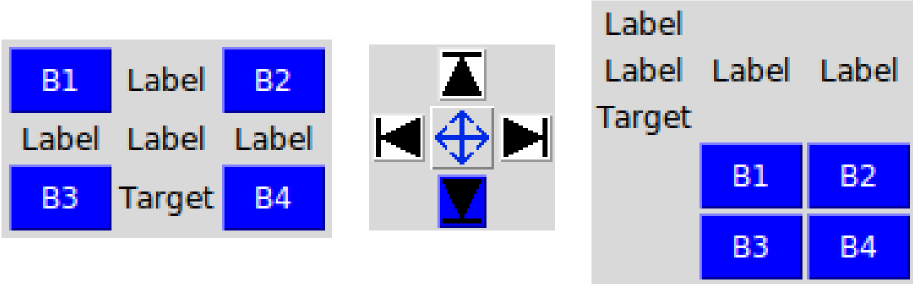

Illustration 47: Moving multiple widgets (left: original design, right: design after moving)

6.6.3 Nested Layouts and the Hierarchy

For the requirements we constituted that either the possibility to nest layouts or to define and insert custom *widgets* is sufficient. However, as custom *widgets* must be layout *widgets* themselves in our implementation, we cannot realize the former without the latter. Therefore, we are now defining the tools to nest layouts.

As before, we waive `Rows` and `Cols` and confine ourselves to employ `Matrices` and add an appropriate entry to the *Palette*. Inserting a `Matrix` works as usual, but inserting into a design, which contains a `Matrix`, differs a little bit. We check whether a `Matrix` is selected and then decide on the basis of its position in the *UI*-tree, where a new *widget* shall be inserted. We identify four cases:

1. A `Matrix` is selected and the target of the insert-operation is this `Matrix`: The new *widget* is inserted into the `Matrix` at (1, 1).
2. A `Matrix` is selected in the same branch as the target (which is in a `Matrix` itself and not the root *widget*): The new *widget* either is inserted into the selected `Matrix` and beside the target or beside its top-most parent in the selected `Matrix`, if the target is on a lower level. The latter case can occur when layouts have been nested multiple times and, in contrast to its children, the right target in the selected `Matrix` may be difficult to access.
3. If no `Matrix` is selected, but the target is in a `Matrix`, the new *widget* is inserted into the root *widget* and beside the target's top-most parent.
4. If the target is in the root *widget*, everything is as before.

Moving in or beside a `Matrix` is similar. But, due to the implementation of the function `createMove` in the module `GUI` and because moving a *widget* from one layout *widget* to another one would have an impact on its reference, moving between different layout *widgets* is not possible at all. However, selected *widgets* may be in different `Matrices` and as well in a different `Matrix` than the target of the move-operation. In the latter case we assume a parent of the target, which is on the same level as a selected *widget*, was actually meant as the target. If such a parent does not exist or if a new target has already been set for another selected *widget* before, the current *widget* is ignored by the operation.

A frequent problem with nested layouts is that such a *widget* is covered by its children and cannot be selected anymore. We could solve this with a simple mechanic, which just selects the parent of a selected *widget*. However, this seems to be impractical, as one would probably prefer to directly select a concrete *widget* and an hierarchical perspective on the design would be useful anyway. Therefore, we design a new component, the *Hierarchy*. This is a list of nodes and entries, where each node corresponds to a `Matrix` and an entry to any other *widget*. Nodes and entries have labels, which consist of a *widget*'s reference name and its kind. Furthermore, nodes can contain nodes and entries themselves and each node has a button to expand or collapse. Due to the limited screen space, this is an important feature again.

We define a new data type, which reflects the model of such a tree view, together with two event handlers for each entry or node

```
data TreeViewModel =
    TVNode String (String -> UIEnv -> IO ()) (String -> UIEnv -> IO ())
        [TreeViewModel]
  | TVEntry String (String -> UIEnv -> IO ()) (String -> UIEnv -> IO ())
```

and a function, which takes a reference for the tree view's root and a `TreeViewModel` and creates a corresponding *widget*:

```
createTreeViewFromModel :: UIRef -> TreeViewModel -> UIWidget
```

If an entry or a node is selected or deselected, its corresponding event handler is called and its label is passed to the handler. The event handler also selects or deselects the *widget* in the *Design View*.

Because the design dynamically changes, the tree view provides operations for these purposes, too. For example

```
addTreeViewFromModel :: String -> UIWidget ->
    TreeViewModel -> UIEnv -> IO ()
```

adds one or more entries or nodes from a `TreeViewModel` to the node with the given label and to the given tree view *widget* and

```
updateEntryOrNode :: UIWidget -> String -> String ->
    UIEnv -> IO ()
```

updates the label of the entry or node with the given label in the given tree view.

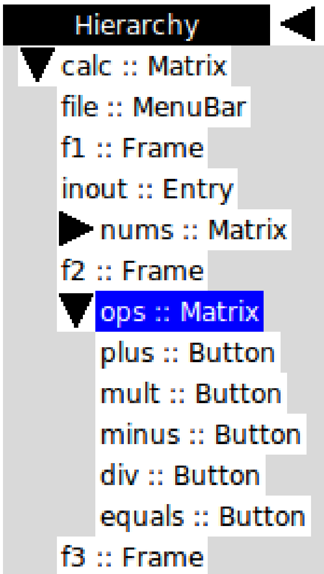

Illustration 48: The hierarchy

Like the *Palette* and the *Properties*, the complete *Hierarchy* can be shown or hidden. Illustration 48 shows the *Hierarchy* for the example design from 6.3.4, where the number-buttons and the buttons for the operations have been put into separate `Matrices`. The node for the former collapsed.

6.6.4 Custom Widgets

In order to separate concerns in a *UI*, *a* more convenient way than using plain nested layouts is to define a custom *widget*, i.e., a *widget*, which consists of several basic ones. These can be reused in the same design, as well as in other designs. The *XML*-files we defined before contain any information, which is required to create a *widget* from it. Hence, any saved design already defines a custom *widget*. We just have to provide the tools to insert these into the current design.

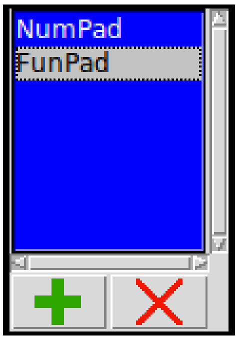

Illustration 49: Custom widgets in the palette

In *FLUID* common *widgets* can be chosen in the *Palette*. A corresponding create-function is then stored in the `PaletteState`. Custom *widgets* fit quite well into this implementation, as they can just provide another create-function, which is performing the necessary operations. However, this function is processing an *XML*-file and applies changes to the `PaletteState`, the `PropertiesState`, the `ReferencesState`, the `ArgumentsState` and the `HandlersState`, as the design, the custom *widget* is derived from, defines. Therefore, it is an IO-function and we have to adjust the `PaletteState` and the other create-functions accordingly:

```
type PaletteState =
   IORef (Maybe (UIRef, UIRef -> UIRef -> UIEnv -> IO UIWidget), [String])
```

As before, the state maybe contains the reference of a selected label in the *Palette* and the corresponding create-function. The function takes a reference for the *widget* to create, a target (for scroll bars) and the environment and returns the resulting *widget*. In order to access custom *widgets*, we add a list box, where *XML*-filenames, without their suffixes, can be added to or removed from, to the *Palette*, see Illustration 49. The new list of strings in the `PaletteState` stores the corresponding paths. Finally,

```
createCustomWidget :: FLUIDState -> [UIRef] -> UIRef -> UIRef -> UIRef ->
    UIEnv -> IO UIWidget
```

creates a custom *widget* (the `FLUIDState`, the prefs and a reference for the list box are applied before the function is stored in the state). It retrieves the current selection from the custom *widget* list box and the path-list and loads the contents of the resulting *XML*-document. Similar to loading a design, it applies the default event handlers (select, insert, …) to the *widgets*, but also merges the properties, reference names, arguments and handlers the document contained, with those of the current design. Because these values may already exist, especially when the same custom *widget* has already been inserted, they are renamed until they are unique (by adding single quotes) and the handlers' arguments are also adapted. As before, the function `insertFromPalette` then creates a *widget* by calling the current create-function in the `PaletteState` and updates the *Hierarchy*.

6.6.5 Copy, Cut and Paste

A useful feature to repeatedly insert and configure equal *widgets*, which also exists in a similar way in many other applications, like word processors, is the option to copy. Hence, it can be seen as a standard we would like to realize.

To copy a *widget* in *FLUID*, its current configuration must be saved to a "clipboard". This covers the actual *widget*, its reference name from the `ReferencesState`, its properties from the `PropertiesState` and its event handlers from the `HandlersState`. Furthermore, similar to moving multiple *widgets*, the positions of the *widgets* to copy must be sorted and saved and, because the *widgets* are selected, their styles from the `DesignState` must be applied to the copies. Because the positions of *widgets* from different levels of the *UI*-tree would be undefined in a compound, just those on the same level as the *widget*, which has been selected at last, are copied. We create a new state, which realizes a clipboard:

```
type ClipboardState = IORef (
     [UIWidget],
     [[(UIRef, (Maybe Int, Maybe Int))]],
     [(UIRef, [Style])],
     [(UIRef, String)],
     [(UIRef, [(String, String, [(String, String)])])]
  )
```

One aspect of the associated function `copyHandler` is to save as much computation time as possible, because the copies may never be used and we want to avoid unnecessary computations. Therefore, some time intensive changes, like the assignment of new references, are skipped until the content of the clipboard is pasted. We neither check where a spanning *widget*'s `NULL` *widgets* are, nor copy the latter at all. Hence, the function is rather simple. We can reuse the function which sorts widgets by their absolute positions and returns those on the same level. Those on other levels are removed from the selection. The states for the selection are retrieved and inserted into the `ClipboardState`.

As soon as `copyHandler` is implemented, cutting is trivial, as the responsible function `cutHandler` just calls `copyHandler` and `deleteHandler`.

In order to insert the content of the clipboard, i.e., to paste, we extend the event handler, which reacts on a right-click on a *widget* in the design. The following operations are then performed:

1. *Widgets* are sorted into a list of rows according to the saved positions of the original ones.
2. New references are created and assigned to *widgets*, as well as new event handlers for selecting, inserting, moving and pasting, which are based on the references.
3. The key-references in the saved lists of reference names, event handlers and properties are replaced by the new ones.
4. `NULL` *widgets* for spanning *widgets* are created and inserted at the correct positions into the list of rows.
5. If the position to paste is above another *widget*, the rows are reversed, i.e., they will be inserted bottom-up.
6. The target, i.e., the *widget*, where the content of the clipboard is inserted relative to, is determined as in 6.6.3.
7. The *widgets* are inserted according to their absolute positions in the rows.
8. In order to create unique names, reference names are renamed like for custom *widgets* and the `ReferencesState`, `HandlersState` and `PropertiesState` are updated.
9. The *Hierarchy* is updated.

Note that the *widgets* are actually not inserted and thus their references stay unbound until the current event handler finishes. This is the reason why we have to perform the operations step by step and cannot, for example, insert `NULL` *widgets* relative to a spanning *widget* and right after `insertWidget` has been called for the latter, as this would suspend the execution due to residuation (see 2.2.6).

6.6.6 Undo and Redo

Another requirement to realize would be the possibility to undo operations, like inserting, moving or deleting *widgets*. Though, a detailed examination of the current implementation has shown that this feature would require many changes of existing functions in the module `FLUID`, especially to undo configurations of a *widget*'s properties. At this point of the development this would be a high risk and would also require much time and testing. Therefore, we do not implement this feature in *FLUID*, but we would like to realize it for the basic *UI*-library, as this will show the power of the *command pattern* the dynamic part of the library is based on.

In chapter 5 we payed attention to cover any relevant information in the command objects, like the state of the *widget* or its parent before the command is processed. Operations like `createDelete` in the module `DynUI2GUI` can be employed to create a command object. In order to undo commands, they must be enqueued before they are processed, for example in a list in an IO-reference. The required information to reverse the command can then easily be accessed. We implement a new function

```
undo :: [ReconfigureItem] -> UIEnv -> IO [ReconfigureItem]
```

for this purpose. It takes a list of `ReconfigureItems`, i.e., command objects, reverses them and returns the result. The following listing shows how an insert-operation is undone:

```
undo ((new, reconf) : rs) env   = do
    cmd <- case reconf of
        GUI.WidgetInsert _ child _ _ _ ->
            case GUI.getRef $ GUI.getConfs child of
                Just r  -> do
                    c <- createDelete (Ref r) env
                    return [c]
                Nothing -> error "DynUI2GUI.undo: Missing reference!"
    cmds <- undo rs env
    return (cmd ++ cmds)
```

The result simply is a delete-command, created by `createDelete`, for the *widget* referred by the reference in the command's child-parameter, which is put into a list.

The other commands are more advanced, as, for example, a *widget*'s original position must be computed for insert- and move-commands. However, these problems can be solved by translating the *widget* from the context of the module `GUI` to the module `UI` and using `findIndexUI` and/or other functions from the module `UI`. Because redoing a command means to undo an undo-operation, the reversed commands are not yet processed here, so they can be put into another queue.

`undo` can now be employed similar to `reconfigure` and calling `reconfigure` on a list of reversed commands performs the corresponding changes. Undoing these realizes redo. The following functions show how this can be generally implemented:

```
undoHandler cmds env = do
    (cs, us) <- readIORef cmds
    if null cs
        then done
        else do
            newus <- undo [head cs] env
            writeIORef cmds ((tail cs), newus ++ us)
            reconfigure newus env

redoHandler cmds env = do
    (cs, us) <- readIORef cmds
    if null us
        then done
        else do
            newcs <- undo [head us] env
            writeIORef cmds (newcs ++ cs, (tail us))
            reconfigure newcs env
```

An IO-reference contains a list of commands and a list of undo-commands. The last command resp. the last undo-command is undone by applying `undo` on it. The result is then enqueued in the list of commands or undo-commands and `reconfigure` is applied.

7 Parsing

At this stage we are able to visually design a user interface and store that in an *XML*-document. Following the architecture we defined in 6.1, the final step is to implement the lower layer, where *Curry* source code is generated from the document.

7.1 XML2Curry

So far, a frequently used approach to manually define a *UI* is to write a single function, which contains the constructors for any of the interface's *widgets*, as well as local declarations for the event handlers. An advantage of this procedure is that one does not have to think about arguments, like references, which have to be available to *widgets* and handlers. On the other hand the source code can be very difficult to read, especially when the *UI* is extensive. Another drawback is that a single *widget* or handler cannot be reused by another module. For the code we generate, we would like to separate concerns from each other.

The idea is to create an own “global” function for any *widget* and event handler. Furthermore, we would like to adapt the quasi-standard architecture for applications with a user interface: *MVC*. If there is a concrete *model*, it is defined by external arguments, which are out of our scope; hence, we stick to the *view* and the *controller*. We further separate the *view* from the *controller* by creating an own module for each. As the *view* should not be manually edited, this approach also has the advantage that this module can just be overridden, when the design has changed.

The following example shows a cutout of the source code for the `ListBox`-*GUI* in 6.4.1:

```
module xml_example where

import UI
import UI2GUI
import xml_example_controller

--- Runs the UI.
main :: IO ()
main = UI2GUI.runUI "XML example" widget

--- The root widget.
widget :: UI2GUI.UIWidget
widget =
  UI2GUI.row
   [UI.Widget
     (UI.Matrix
       [[common_list_box_listbox listbox
        ,scroll_bar_vertical_a_b a_b listbox]
       ,[button_deleteButton deleteButton listbox,null_a_d a_d]])
     Nothing (Just a) [] [UI.Class [UI.Bg UI.Default,UI.Fg UI.Black]] []]
  where a free
        listbox free
        a_b free
        deleteButton free
        a_d free
```

We left the function definitions for the main *widget*'s children out, as they are similar to `widget`, but without the local declarations of free variables for references and the surrounding `Row`. The latter is required, because the *GUI*-implementation neglects the main *widget*'s configuration.

We derive the general structure of the *view* module: The name of the module and a list of imports must be declared. The module `UI` must always be imported, as we use its symbols and types, like the kind of a widget, as well as styles and colors. The other imports cover the type of the *UI*, i.e., an abstract *UI*, a *GUI* or a *WUI*, which is determined by a user's choice and the import of the *controller* module.

The imports are followed by the function declarations and definitions. The function `main` is optional, as the *view* may just be part of a larger *UI*, but if it is generated, it also requires the main window's title. The example does not cover external arguments, which can be defined for the main *widget* in *FLUID*. If there are any, we declare free variables for them in `main`, in order to avoid compiler errors.

The remaining functions form a flat structure, i.e., a list. The name of such a function is a combination of the kind of *widget* it creates (the main *widget* excepted) and its reference name. Each one consists of a type declaration and a rule, which defines the function. The right-hand side of the definition may contain several calls of other *widget* creating functions, which is a hierarchical structure again and the main *widget* also locally declares free variables for any *widget*.

The *controller* module for the example is trivial, as there is just a single event handler and we just declare these and do not define them, but the following code snippet may clarify the idea:

```
module xml_example_controller where

import UI2GUI

deleteHandler :: UI2GUI.UIRef -> UI2GUI.UIEnv -> IO ()
deleteHandler listbox env = done
```

In order to realize the concept, we create a new module `XML2Curry` and the function

```
xml2curry :: Bool -> String -> String -> String -> Bool -> String ->
    XmlExp -> IO ()
```

which requires a parameter whether a `main` shall be generated and the title. Furthermore, it requires the name of the module to base the *UI* on (`UI`, `UI2GUI` or `UI2HTML`), a filename for the *view*, a parameter whether a *controller* shall be generated and a filename for it and finally an expression containing the content of the *XML*-document.

In order to generate the source code we do not have to directly parse the `XmlExp` to a string, as this approach would be error prone, but instead use the modules `AbstractCurry` and `AbstractCurryGoodies` to create a valid *Curry* program. The latter module contains abbreviations for frequently used constructs. We delegate the pretty printing of the resulting abstract program to `PrettyAbstract`. Thus, we model the code structure described above with the following `AbstractCurry` type constructors:

- A `CurryProg` describes one module. Its relevant parts are the name of the module, a list of import declarations and a list of function declarations.
- A function declaration `CFunc` consists of a function name, as well as its arity and visibility, which is always `Public` here, i.e., the generated module exports any function. Furthermore, it covers a type expression and a list of rules. Note that the type expression is not optional; hence, using type inference is not possible. This is the main reason why we introduced types for external arguments and references in *FLUID* at all. It is not possible, for the arguments at least, to guess their types without a concrete definition for an event handler using the argument (and not just a stub), while parsing the *XML*-document. A function declaration may be complemented with a comment by using `CmtFunc` instead of `CFunc`.

- A type expression `CTypeExpr` is either a type variable, a type constructor with a qualified name, e.g., `("Prelude", "Int")` and a list of type expressions for the constructor's arguments or a type expression for a function, which is just a composition of type expressions.
- A rule `CRule` contains a list of patterns, corresponding to the function's left-hand side, a list of pairs of expressions, one for a guard and one for the right-hand side and finally a list of local declarations. In our case there is just a single pattern for each function, which has no guard and local declarations are solely required for the reference declarations in the main *widget*.

Parsing an *XML*-document containing a *UI*-definition in order to set up a `CurryProg` is not so straightforward as it may seem in the first place. What's problematic here is their different structures. The document is strict hierarchical, while the program is flat on the one hand and has a hierarchical component, representing a parent-child relationship, on the other hand. A function's type is derived from its arguments. The arguments are derived from the *widget*'s reference name, as well as its event handlers, resp. their arguments and those of its children. Therefore, we have to perform a bottom-up analysis of the *UI*-definition and return any relevant information, i.e., the function name, the arguments, the event handlers' signatures and the expression, which is actually creating the *widget*, from a *widget* to its parent, until we arrive at the main widget and can finally create the function declarations. `xml2curry` then creates a `CurryProg` for the *view* and another one for the *controller* (if desired) and writes them to the given files.

As we would like to integrate the source code generation into *FLUID*, we design a dialog matching the signature of `xml2curry`, see Illustration 50. We can already design it in *FLUID* and generate the code by manually calling `xml2curry`; we just have to define the event handlers. This is very easy, as we basically take the current values of the text fields, the check buttons and the menu and pass these to `xml2curry`. As we create a `main`, the dialog can either run as a standalone application or the main *widget* can be directly used in *FLUID*.

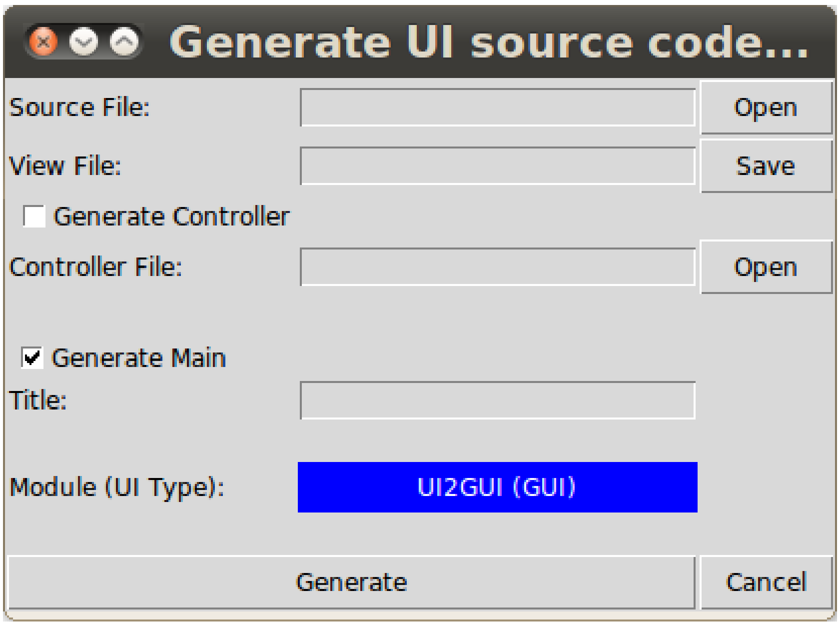

Illustration 50: UI generator dialog

7.2 Curry2XML

We originally planned to implement the other direction, i.e., *Curry* source code to *XML*-files, too. But it would be very difficult to cover any variant a *GUI* can be implemented of. Thus, a solution would probably only consider a few of these variants. As this would not be very useful and because we had rather realized as much of the requirements as possible, we waive implementing this module.

8 Conclusion

We created an application, which aids developers with the challenges of designing a user interface.

We took advantage of the abstract *UI*-library, which allows to define *GUIs* and *WUIs*. Extensions of the library especially allow to dynamically insert, delete, move and configure *widgets* at runtime. The current state of the *UI* can also be retrieved from a global *UI*-definition, which is automatically updated in the background. Furthermore, operations to search and update *widgets* in this tree-structure have been implemented.

The extensions of the library should be downward compatible, as we payed attention to limit the changes of existing code to just a few lines and rather extended instead of changed.

As *FLUID* is a desktop application itself and thus is based on the *GUI*-part of the *UI*-library and the appearance of a *GUI* differs a little bit from a *WUI*'s, the *WYSIWYG* principle cannot be applied in all respects. However, we implemented workarounds where possible and even properties, which are not applicable to a *GUI*, can be applied to a design. Furthermore, there are different ways to achieve a goal. For example, in order to replicate *widgets*, they can be copied and pasted somewhere else (also to another design) or they can be saved as a custom *widget* and then imported to any design.

Most functional requirements, which have been derived from popular *GUI-builders*, have been realized. We covered any of the major and most of the minor requirements except for drag and drop, undo and the concurrent configuration of properties of multiple *widgets*. For the most part, the optional ones have also been implemented, but, due to restrictions, direct manipulation of *widgets* could just be realized to some degree.

The *layered model FLUID*'s architecture is based on and especially the intermediate *XML*-files, on the one hand, allowed to independently develop the layers from each other and, on the other hand, layers can be substituted or extended by new ones. Hence, for example a source code generation layer, which produces code for other languages than *Curry*, could be realized without breaking the other layers. Implementing new layers could be subject to future work.

Bibliography

[Hanus 2000] M. Hanus (2000). A Functional Logic Programming Approach to Graphical User Interfaces. In Proc. of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00), pages 47-62. Springer-Verlag.

[Hanus, Kluß 2009] M. Hanus, C. Kluß (2009). Declarative Programming of User Interfaces (diploma thesis). In Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL'09), pages 16-30. Springer-Verlag.

[Wikipedia] Wikipedia (english). Available at `http://en.wikipedia.org` (accessed in December 2010).

[Molin et al. 1996] P. Molin, F. Ström (1996). A GUI Builder for Erlang/GS (diploma thesis). Uppsala University, Uppsala, Sweden.

[Curry] Curry - A Truly Integrated Functional Logic Language. Available at `http://www.curry-language.org/` (accessed in December 2010).

[Freeman et al. 2008] E. Freeman, E. Freeman, K. Sierra, B. Bates (2008). Entwurfsmuster von Kopf bis Fuß, orig. Head First Design Patterns, pages 191-233, 528-544. O'Reilly.

[Curry tutorial] S. Antoy, M. Hanus. Curry - A Tutorial Introduction. Available at `http://www-ps.informatik.uni-kiel.de/currywiki/documentation/tutorial` (accessed in January 2011).

[Tk tutorial] Tk tutorial. Available at `http://www.tkdocs.com/tutorial/index.html` (accessed in January 2011).

[NetBeans] NetBeans IDE (v6.9). Available at `http://www.netbeans.org` (accessed in August 2010).

[Visual Studio] Microsoft Visual Studio 2010 Express (v10.0). Available at `http://www.microsoft.com/express/` (accessed in August 2010).

[GUI Builder] GUI Builder (v2.5). Available at `http://spectcl.sourceforge.net/` (accessed in August 2010).

[#develop] SharpDevelop (v3.2). Available at `http://www.icsharpcode.net/OpenSource/SD/` (accessed in August 2010).

[SWEBOK 2004] The Institute of Electrical and Electronics Engineers (IEEE) (2004). Guide to the Software Engineering Body of Knowledge. IEEE. Available at `http://www.computer.org/portal/web/swebok/` (accessed in August 2010).

[Sommerville 2007] I. Sommerville (2007). Software Engineering 8, pages 242-265. Pearson Education Limited.

A Format of the XML-Files

The following *document type definition* (*DTD*) describes the structure of the *XML*-files for *FLUID* proposed in 6.4. Note that some values of the attribute `kind` are not implemented in *FLUID*. For `Widget` these are `list_box`, `list_box_item`, `name`, `link` and `radio_button` and `name_value` for `Style`. Also note that the *DTD* is not explicitly associated with the generated documents.

```
<!ELEMENT Widget (Argument | Style | Widget | Row)*>
<!ATTLIST Widget
  kind      (matrix | row | col | label | button | entry | text_edit |
             scale | check_button | menu_bar | menu | menu_item |
             menu_separator | canvas | list_box | list_box_item |
             common_list_box | name | link | radio_button | null |
             scroll_bar_horizontal | scroll_bar_vertical)
                          #REQUIRED
  refname   ID            #REQUIRED
  label     CDATA         #REQUIRED
  height    CDATA         #IMPLIED  <!--Required for text_edit, canvas
                                        and list_box-->
  width     CDATA         #IMPLIED  <!--Required for text_edit and
                                        canvas-->
  min       CDATA         #IMPLIED  <!--Required for scale-->
  max       CDATA         #IMPLIED  <!--Required for scale-->
  checked   (True | False) #IMPLIED  <!--Required for check_button and
                                        radio_button-->
  selection CDATA         #IMPLIED  <!--Required for list_box-->
  value     CDATA         #IMPLIED  <!--Required for list_box_item and
                                        name-->
  selected  (True | False) #IMPLIED  <!--Required for list_box_item-->
  target    IDREF         #IMPLIED  <!--Required for
                                        scroll_bar_horizontal and
                                        scroll_bar_vertical-->
>

<!ELEMENT Argument EMPTY>
<!ATTLIST Argument
  name CDATA #REQUIRED
  type CDATA #REQUIRED
>

<!ELEMENT Row (Widget)*>

<!ELEMENT Style (Item)*>
<!ATTLIST Style
  kind (align | text_align | text_color | fill | height | width |
        active | foreground | background | font | border | display |
        name_value | col_span | row_span | list_box_list | image)
             #REQUIRED
  position (center | left | right | top | bottom)
             #IMPLIED <!--Required for align and text_align-->
  color (default | black | blue | brown | cyan | gold | gray | green |
         magenta | navy | orange | pink | purple | red | tomato |
         turquoise | violet | white | yellow)
             #IMPLIED <!--Required for text_color, foreground and
                          background-->
  direction (x | y | both)
             #IMPLIED <!--Required for fill-->
```

```
  value CDATA #IMPLIED <!--Required for height, width, active, display,
                         name_value, col_span and row_span-->
  style (bold | italic | underline | dotted | dashed | solid)
            #IMPLIED <!--Required for font and border—>
  name CDATA  #IMPLIED <!--Required for name_value-->
  path CDATA  #IMPLIED <!--Required for image-->
>

<!ELEMENT Item EMPTY>
<!ATTLIST Item
  value CDATA #REQUIRED
>

<!ELEMENT Handler (Argument)>
<!ATTLIST Handler
  event   (DefaultEvent | FocusOut | FocusIn | MouseButton1 |
           MouseButton2 | MouseButton3 | KeyPress | Return | Change |
           Click | DoubleClick)
                #REQUIRED
  command CDATA #REQUIRED
>
```

B Contents of the CD-ROM

source/

- **AbstractCurryGoodies.curry –** Abbreviations for AbstractCurry types, which are frequently used.
- **DynUI2GUI.curry –** Library to dynamically insert, remove, configure and move *widgets* at runtime.
- **FLUID.curry –** The *FLUID UI-builder*.
- **FLUID2XML.curry –** Library to translate *widgets* and states from *FLUID* to *XML*.
- **GUI.curry –** The *GUI*-library.
- **GUI2UI.curry –** Library to map *widgets* from the *GUI*- to the *UI*-library.
- **UI.curry –** Main module of the *UI*-library.
- **UI2GUI.curry –** Sub-module of the *UI*-library to create a *GUI* from a *UI*-definition.
- **UI2HTML.curry –** Sub-module of the *UI*-library to create a *WUI* from a *UI*-definition.
- **UIGenerator.curry –** *GUI*, which creates *Curry* source code from *XML*-files.
- **UIGeneratorController.curry –** Controller for the UIGenerator-*GUI*.
- **UIWidgets.curry –** Combined *widgets*, like a tree view or a choice, for the *UI*-library.
- **XML2Curry.curry –** Library to translate *XML*-files to *Curry* source code.
- **XML2FLUID.curry –** Library to translate *XML*-files to *widgets* and states for *FLUID*.
- **docs/**
  - **about.txt –** The about for *FLUID*.
  - **help.txt –** The help for *FLUID*.
- **images/**
  - Images for icons.

documentation/

- **thesis.pdf –** This document.

C Installation of the Software

The software has been tested on Ubuntu Linux, Lucid and Maverick, using SWI-Prolog v5.8.0, *Tcl/Tk* v8.5 and the *PAKCS* Linux binaries v1.9.2.

Prerequisites

- A Prolog implementation must be installed. If SWI-Prolog is used, the library "swi-prolog-clib" may be required, which is currently unavailable to the most recent version of SWI-Prolog, i.e., an earlier version may be used.
- *Tcl* and *Tk* v8.5 with the Wish command line interpreter. Earlier versions should still work, but there may be drawbacks regarding the software's appearance.
- An installation of *PAKCS*. The configuration file ".pakcsrc" should exist in the home-directory. In this file, the key "libraries" should at least contain the path to the folder `/tools/ui` in *PAKCS*' installation directory, e.g.,

  ```
  libraries=<pakcshome>/tools/ui
  ```

  where `<pakcshome>` is the installation directory of *PAKCS*.

Installation

- Copy the folder "source" from the CD to a writable location.
- Run *PAKCS* in a terminal.
- Compile the modules in the copied folder by typing

  ```
  :cd <copyfolder>/source
  ```

  where `<copyfolder>` is the folder where "source" has been copied to and

  ```
  :l FLUID
  ```

- Run *FLUID* by typing

  ```
  main
  ```

D Manual for the Software

The following manual for *FLUID* can also be directly accessed in the menu *?* of the application.

Contents:

(1) Create a New Design

A new design can be created by choosing 'New' in the menu 'File' or via the file-button in the tool bar. The current design can be saved before the change applies. Note that you may have to select a widget in order to update the properties for the new design.

(2) Load or Save a Design

To load a design choose 'Open' in the menu 'File' or press the folder-button in the tool bar. The current design can be saved before. Note that you may have to select a widget in order to update the properties for the loaded design.

Analogue, to save a design, choose 'Save' from the menu 'File' or press the disk-button in the tool bar.

(3) Choose and Insert Widgets into a Design

A widget to insert can be selected in the component labeled 'Palette' by left- clicking the corresponding image. More widgets can be chosen by turning the current page via the arrows below the palette.

If the design has just been created, the widget can be inserted by right- clicking into the white space in the center. If there already are widgets in the design, the desired location of the new widget relative to an existing one can be chosen in the menu 'Edit' or with the arrow-buttons in the tool bar. Right-clicking a widget will then insert the new one, if the insert-mode is active. The mode can be changed with the option 'Insert/Move/Paste' in the menu 'Edit' or via the button in the center of the arrow-buttons in the tool bar. The plus-icon indicates that the insert-mode is active, the multi-arrow-icon that the move-mode is active and the clipboard-icon (with an arrow pointing away from the icon) that the paste mode is.

In order to insert into a matrix, it must be the widget, which has been selected at last (see also next section). Right-clicking any of its children or their children inserts into the selected matrix and relative to the former, if the insert-mode is active.

The palette can be faded in and out with the arrow next to its label.

(4) Select and Arrange Widgets in a Design

A widget in the design can be selected by a left-click. This will change the widget's appearance. (To select a menu bar, left-click it and then one of its menu items.) Multiple widgets can be selected by repeating the procedure. The current selection can be deselected by left-clicking one of the widgets again or the empty space below the design. If no other widget is selected, the main widget is.

Another way to select a widget is by left-clicking its entry in the hierarchy. The hierarchy is the component to the left of the palette, which is hidden by default. It is especially useful to select matrices, which are covered by other widgets. A node in the hierarchy can be expanded or collapsed by clicking on the arrow-icon next to the node.

The selection can be deleted by clicking the cross-button in the tool bar or via the menu 'Edit' and the option 'Delete'.

One or more selected widgets can be moved by choosing the desired position relative to another one in the menu 'Edit' or the tool bar and right-clicking a widget in the design, while the move-mode is active (see 3). If multiple widgets shall be moved, they may be spread over the design, but will merge at the targeted position. Note that widgets cannot be moved between different matrices.

(5) Define a Widget's Properties

To configure a widget, it must be selected. Just the widget which has been selected at last can be configured.

In contrast to the palette, the component labeled 'Properties' does not contain several pages, but sections and each one can be separately shown or hidden by clicking the arrow-icon next to a heading. Note that the less sections are visible, the faster the selection of a widget is.

The properties can be faded in and out, as well.

(5.1) Configure Simple Properties (Text, Layout and Appearance)

Simple properties, like a widget's label, text align and background color, can be directly applied by navigating to the corresponding section and typing the desired text in, followed by 'Return' or by choosing the value in a drop-down menu.

(5.2) Configure a List Box

The content of a list box can be configured in a separate section labeled 'Special' in the properties. To add an item to the box just type in the value and press 'Return' or first select an existing item to add behind that. To update an existing item select it, type in the new value and press the tick-button. A selected item can be deleted with the cross-button.

(5.3) Configure a Menu Bar

Similar to a list box, a menu bar's content can be configured. One can define the main menu on the left and a (direct) sub-menu on the right. So far, just two levels of menus can be configured. In addition to the procedure described in 5.2, the type of an item has to be chosen in a drop-down menu. A sub-menu can only be configured if the selected item in the main menu is of type 'Menu'.

(5.4) Define a Reference and Arguments

A widget is identified by its reference. The reference will be generated, but can also be customized in the section labeled 'General' in the properties. Note that a reference must be unique. If the main widget is selected, the application's arguments can also be defined. Behind the label 'Name' and 'Type' a new one can be defined with the given name and a qualified type, e.g.,

name = db

type = DBModule.Database

Existing arguments can be browsed or deleted above.

Any references and arguments are especially relevant and available to event handlers and renaming resp. deleting one of these values leads to corresponding changes of the handlers' arguments (see below).

(5.5) Define Event Handlers

Below the reference or arguments in the properties, a widget's event handlers can be defined. Pressing the plus-button creates a new event handler. The cross-button deletes a selected handler. The structure of an event handler is the event, followed by a colon and the command (the name of the function to call on the event) and then a list of arguments for the command in parenthesis. To configure a handler, select it and choose its event from the drop-down menu. Type in the command name and press 'Return' or choose an available argument to add and press the tick-button or an existing one to delete and press the cross-button. Available arguments are arguments defined for the main widget, as well as any references.

There are two ways to define event handlers for a menu bar's items:

If the selected widget is a menu bar, an event handler can be denoted to be used for a menu item by adding a prefix to the command. The prefix may be a sequence of indices in brackets ('[]'), where each index denotes the menu item's position (starting with zero) in the main menu and sub-menus accordingly. For example

[3][7]doSomething

would assign the handler with the command 'doSomething' to the fourth entry in the main menu (which should be of type menu) and the eighth in that sub-menu.

The second and most likely preferable way to define a menu item's handler is to do that in the section 'Menu Handlers' in the 'Special'-section of a menu bar's properties. For a selected menu item, the procedure is exactly as for a common widget described above, but indices are automatically added.

(6) Copy, Cut and Paste

Any selected widget, together with their properties, can be copied or cut to the clipboard. In order to copy the selection, choose 'Copy' in the menu 'Edit' or click on the clipboard-button (with an arrow pointing to the icon) in the tool bar and in order to cut, choose 'Cut' in the menu or click on the scissors-button in the tool bar. To paste previously copied or cut widgets, choose the paste-mode and insert them by a right-click on a widget, as described in (3). Note that the references and corresponding arguments of event handlers of these widgets may be renamed, if they would not be unique. Also note that widgets from different matrices cannot be copied together.

(7) Custom Widgets

Any saved design also defines a custom widget. It can be imported on the second page of the palette by clicking the plus-button there and opening the corresponding file. It may then be chosen in the list above and used as any other widget in the palette. A custom widget can be removed via the cross-button in the palette. Note that, as also described in (6), references and arguments of event handlers may be renamed.

(8) Generate Source Code

In the menu 'File' a dialog to generate source code from a previously saved design can be opened with the option 'Generate'. Choose the design behind 'Source File' and the file where the source code shall be generated behind 'View File'. A 'Controller File', which contains event handler stubs, can either be generated or an existing one opened. If the generated UI shall run as an application, a main function with a title for the window can be specified. The choice behind 'Module' denotes which type the UI shall be of, i.e., whether be it a GUI, a WUI or an abstract UI-definition. A click on 'Generate' generates the source code. Note that existing files will be overridden.

When generation finished, the dialog closes.