UNIVERSITEIT ANTWERPEN
Departement Wiskunde en Informatica

2003-2004

Glib-C: C as an alternative Object Oriented Environment

Steven Hendrickx

Proefschrift ingediend tot het behalen van
de graad Licentiaat in de Wetenschappen

Begeleider:	Filip Van Rysselberghe
Promotor:	Prof. Dr. S. Demeyer
Copromotor:	Prof. Dr. B. Watson

Samenvatting

De programmeertaal C mag dan wel geen Object Georiënteerde Programmeer Taal (OOPL[1]) zijn, C laat de ontwikkelaar wel toe om een object georiënteerde (OO[2]) programmeer techniek toe te passen.

Een typische OOPL biedt een abstractie aan voor het onderliggende object model en de bijhorende mechanismen. Wanneer men OO gaat programmeren in C, gaat men de mechanismen toepassen die in een OOPL achter een abstractie verborgen liggen. B.v. de eerste C++ compilers waren amper C++ naar C vertalers. Het zijn dan ook vooral de technieken en ideeën van het C++ object model die men toepast in C.

Hoewel men OO kan programmeren in C, is dit over het algemeen niet aangeraden. Het is duidelijk dat door het ontbreken van een abstractie van het object model er extra moeilijkheden opduiken. Bjarne Stroustrup, de vader van C++, stelt dat het onnodig moeilijk is om OO te programmeren in een taal die daar geen constructies voor aanbiedt.

Ondanks deze "waarschuwing" wordt OO programmeren in C wel toegepast. De techniek van OO programmeren die we in deze thesis bespreken is ontstaan in de open source cross-platform user interface library GTK+, en wordt in tal van andere open source projecten gebruikt.

Deze projecten maken allemaal gebruik van de C library Glib. Dit is een cross platform hulp library die net als de C++ standaard library STL, abstracties aanbiedt voor datatypes in de aard van "vector" en "list". Daarnaast biedt de Glib library ook een type en object systeem aan, dat de ontwikkelaar toelaat om een OO ontwerp te implementeren in C.

In deze thesis spreken we echter niet over OO programeren in C, gebruik makend van de Glib library, maar over Glib-C. Glib-C is gedefinieerd in deze

[1]OOPL: Object Oriented Programming Language
[2]OO: Object Oriented

thesis als OO programmeren gebruik makend van het object en type systeem van Glib, maar waar ook een verzameling van idiomen zijn gerespecteerd. Idiomen zijn een verzameling van regels die bepalen hoe een bepaalde taal moet worden "gesproken". Idiomen bepalen een specifieke stijl. In het geval van Glib-C moeten er een aantal elementen aanwezig zijn, en zij moeten er volgens bepaalde regels uitzien. M.a.w. idiomen komen vaak neer op een verzameling van naamconventies.

De vraag is natuurlijk: met Stroustrups waarschuwing in het achterhoofd, is Glib-C wel een waardig object georiënteerd alternatief? Ja, het laat ons toe een OO ontwerp te implementeren. Maar, moeten ontwikkelaars het wel doen als het extra veel moeite kost t.o.v. b.v. C++? Kunnen ontwikkelaars moderne object georinteerde ideeën toepassen zoals het "groeien van software"?

Vooreerst we deze vraag proberen te antwoorden 2 opmerkingen: Ten eerste, Frederick P. Brooks Jr. stelt in zijn artikel "The Silver Bullet" dat een OOPL niet de hoop heeft om software ontwikkeling revolutionair te verbeteren, maar dat wel de ideeën van OO belangrijk zijn.

Ten tweede, Bjarne Stroustrup kijkt naar de taal in puur isolement. Dit is eerder een naïeve en kortzichtige kijk vermits een taal niet een alleenstaand element is, maar slechts een element in een groter gehele, met name de hele toolsuite. Welk element bepaalde verantwoordelijkheden draagt, of bepaalde abstracties biedt, doet er niet toe. Belangrijk is dat uiteindelijk de toolsuite een productieve en comfortabele gebruikerservaring biedt.

Daarom nemen we als criteria voor een geschikt object georiënteerd alternatief dat de omgeving een bepaald niveau van comfort moet aanbieden en tegelijkertijd de ontwikkelaar toe laat om software te groeien.

Daarom proberen we in deze thesis toolsupport te bouwen voor prototyping/code generatie, testing en refactoring in Glib-C. Deze mogen als de basis van "software groeien" beschouwd worden. Aan de hand van deze ervaringen proberen we een antwoord te geven op de vraag: is (toolsupported) Glib-C een geschikt object georiënteerd alternatief?

Telkens kijken we naar wat we nodig hebben om succesvol een bepaalde tool te bouwen. Het is duidelijk dat tools informatie moeten kunnen extraheren van de Glib-C code en bibliotheken. Hiervoor steunen wij vooral op de idiomen. Vervolgens gaan we de benodigde idiomen kritisch bekijken om zo een verbeterde/minder strikte vorm van idiomen te verkrijgen.

Glib-C kan uiteindelijk toolsupported worden, wat op zijn beurt een niveau van comfort aanbiedt en het groeien van software toelaat. Glib-C als een tool op zich is een moeilijke omgeving. Maar in een realistische omgeving waarbij de gehele toolsuite wordt bekeken als één entiteit, biedt Glib-C een waardig OO alternatief aan. Dit betekent niet dat als men Java (b.v. in combinatie met de Eclipse IDE) wilt gebruiken voor een bepaald project, dat Glib-C even geschikt is. Qua OO, zijn beide omgeving zeer sterk omdat beiden omgevingen werken met dezelfde benodigde OO concepten en dat zo ons toelaat er op een goede manier mee te werken. Doch zijn Glib-C en Java zo verschillend dat er andere dingen in rekening gebracht moeten worden. Glib-C zal b.v. meer geschikt zijn voor platform libraries, terwijl Java meer geschikt is om eind gebruikers toepassingen te schrijven.

Abstract

Object Oriented (OO) programming in C is often being diminished, although it is being applied today. However its viability as object oriented alternative has never been seriously investigated. Therefore we evaluate OO programming in C by looking at C code in which idiomatic and systematically OO programming is applied by means of Glib-C. We especially evaluate whether this approach allows us to grow software by looking at modern OO techniques in the context of growing software like prototyping, testing and refactoring.

Acknowledgments

I'd like to thank all the people who made it possible to write me this thesis. Foremost I want to thank my promoter Serge Demeyer for spiking my interest in software development and engineering. I also want to thank LORE assistant Filip Van Rysselberghe whose tips, critical reviews and guidelines were indispensable to me.

Of course thanks to my parents who made it possible for me to study at this university and whose patience was sometimes indispensable to handle my temper.

Finally, I want to thank my colleagues, friends and proof readers for helping out and supporting by whatever the smallest bit. A special thanks goes out to Kelly Casal Mosteiro, Benny Van Aerschot, Bart Van Rompaey, Oliver Celis Salazar and Phan Than Vang.

Contents

Chapter 1

Introduction

1.1 About

Object Oriented (OO) Programming in the programming language C is possible, although C is not an Object Oriented Programming Language (OOPL). However, while C enables the developer to use an OO style programming technique, it is generally not advised: OO programming in C is cumbersome and requires excessive use of boiler plate code, affecting e.g maintainability and productivity. However, OO programming in C is traditionally judged from looking at the language in pure isolation.

In this thesis we look at OO programming in C from the point of view of the tool-suite: the language is only a mere tool in such a suite. We will investigate the OO power of such a tool-suite, which incorporates OO in C. We especially look if this solution allows us the grow software.

For this we take a look at the Glib library. This library provides a type and object system for C, enabling the C developer to express his or her OO designs into C. OO programming in C using the Glib library, is widely used in the open source world.

1.2 Thesis Structure

In chapter 2 we introduce Object Oriented programming in C. First however, we try to tackle the question "what is object orientation?". As in many publications we don't want to be dogmatic but want to try to capture the essence of OO as it is seen in general. In that chapter we also try to touch some of the ideas that will be used in this thesis.

In chapter 3 we introduce Glib-C. Basically this is C code using the type and object system of Glib and conforms to a set of idioms. These idioms are of utmost importance as they are the cornerstone of Glib-C code: Glib-C code can only be ”spoken” if this set of idioms is respected.

In chapter 4 we ask ourselves the question ”what is the OO power of Glib-C?”. There we summarize the criteria we base ourselves on to determine if an environment is a viable OO platform. In this chapter the problems with Glib-C are outlined and a description to a run to the solution is given. Basically, this run to the solution consists of trying to make tool support, to support the idea of growing software. Depending on the requirements and the success of these tools, we try to judge on the OO power of Glib-C.

In the next 3 chapters (chapters 5, 6 and 7), these tools and our experiences with them are outlined. Chapter 5 is about prototyping and code generation. Chapter 6 about testing and finally, chapter 7 is about refactoring. It are these topics that indeed are essential in the idea of growing software.

In chapter 8 an overview of the lessons learned is provided and we draw a conclusion on the question whether C is a viable alternative OO environment.

Chapter 2

Object Orientation and C

2.1 What is Object Orientation

2.1.1 Introduction

Object Oriented Programming (OOP) has its roots as early as in the sixties. In that period the Simula languages became available: those were Simula-1 and, most notably, Simula-67 which appeared in 1967. The latter was the first programming language to have the concepts that many modern Object Oriented (OO) developers are familiar with today: encapsulation, objects, classes, subclasses (inheritance) and virtual procedures (polymorph-ism).

Nevertheless Object Orientation is more than just a way of programming: it is a way of thinking, it is a paradigm. In this paradigm we think in terms of objects. A representation of real world entities with encapsulated data. Objects sent messages to each other and manipulation of such an object only happens through such messages. Those are known via a well defined interface. In the Object Oriented paradigm we identify such objects and try to distribute the responsibilities evenly among those objects. Besides that, relations between those objects, like inheritance and association, are identified.

However, trying to define OO exactly is very difficult: it is some sort of trap. What definition you come up with, there are always people who think the definition is too vague, incomplete or describes properties that do not belong there. In history, OO has been many times described before, trying to formally define what OO is. This suggests that maybe OO cannot be defined exactly. Nevertheless, when asked what OO is, mostly a list of criteria is sum up and conforms more or less with the concepts that were available in Simula-67.

2.1.2 An overview of concepts

Let's take a look at the common concepts that are often associated with OO. Encapsulation, objects, classes, inheritance and polymorph-ism may be regarded as the essential basis for OO.

For one, encapsulation may as well be the corner stone of OO. [PJ00] summarizes this nicely by his story about putting a group of OO gurus together who have to create a list of what they see as properties of OO. The list contained only the concepts that all the gurus had in common on their list. This resulted in only one word: "encapsulation".

Moreover, [PJ00] sums 9 concepts which are "encapsulation", "information and implementation hiding", "state retention", "object identity", "messages", "classes", "inheritance", "polymorph-ism" and "genericity". We group most of those concepts together to the five we stated (as will be noted in our overview).

Also, Stroutstrup asserts "The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time)". [Str91]

Indeed, it are those concepts that also return in many environments like C++, Java, Smalltalk, .Net and other class based OO environments.

Although the reader should be familiar with those concepts, lets take a closer look at them for sake of clarity and completeness in context of this thesis. The given description of those concepts is by no means comprehensive or complete, but that's not the intention.

encapsulation Encapsulation and data hiding are important concepts used in OO. With encapsulation data is closed in an object where the user is provided with a well defined interface such that the data only can be manipulated through that interface. With data hiding, the data is not only encapsulated (think "protected") but also invisible from the user.

objects and classes An object is described by a class, where the class describes a type. An object is an instance of such a type. Often an object is being summarized as a state, an identity and behavior [Boo91]. The state of an object is determined by the values of its encapsulated data. The Behavior of an object is defined by the interface on that object. An identity can be any identifier, often referred to as an object handle or reference.

inheritance Inheritance is a concept in which a type can be extended: a type can be specialized. It is said that a class B extends another class A and thereby inherits all the properties from class A. I.e. it inherits both data and interface from class A. Inheritance is an important relation between classes. This relation is described by an inheritance tree.

polymorph-ism Polymorph-ism actually means several things, like message name overloading and operator overloading. However, in the context of this thesis a very strict meaning to polymorph-ism is given. Polymorph-ism in the context of traditional OOPLs is the possibility of an object to take different forms at runtime: sending a message to an object of a certain (static) type can invoke different behaviors depending on the runtime (or dynamic) type.

But even this list is of course subject to discussion. A discussion which has been many times done before. Some people require that some other concepts should be part of the OO paradigm, like exception handling, garbage collecting, built-in operator overloading, runtime introspection and reflection.

However, the 5 latter concepts are tight to the programming language/environment itself. When OO is viewed purely as a way of thinking, we find ourselves on the level of program analysis and design. The 5 latter concepts are not important on this level. What remains are the notion of classes and the relation between them. In this way OO becomes primarily what is known as Object Oriented Analysis and Design (OOA&D).

Again, for the sake of clarity, the latter 5 concepts are described here:

exception handling A mechanism for anticipating and handling exceptions. An exception occurs when something does not go as was intended. A mechanism for easy exception handling allows the developer to handle such an exception elegantly. The most known exception handling systems are those in C++, Java and C#.

garbage collecting A system that frees unused memory in the running program. This frees the developer of handling memory management manually.

built-in operator overloading The ability to override the meaning of existing operators such as "+" when used in the context of certain objects.

runtime introspection The ability to introspect an object at runtime. This enables the developer to query the object and ask for runtime type information (RTI or RTTI). E.g. what type are you? What is your name? What is your parent in the inheritance tree?

reflection Reflection is defined by [MIKC92] as "the ability of an executing system of programmed objects to make attributes of the objects to be themselves the subject of computation". Both Java and .Net are said to have reflection. We make a distinction between two levels of reflection. The first one is as we know it in Java: there is runtime introspection and byte-code introspection. The second one can be summarized as a system where runtime introspection, byte-code introspection and runtime byte-code generation is available such as in the .Net platform. A level 0 of reflection may be regarded as runtime introspection.

2.1.3 The essence

Up till now we have spoken about OO in different forms. Indeed, OO compromises different aspects, like the things we have mentioned before: programming, analysis and design. However, what is important is the paradigm surrounding OO and all its different aspects.

When dealing with those different aspects, it is important to know that all those aspects are actually loosely coupled. Nevertheless they all rely on the same paradigm. This can be summarized as:

- A design can be Object Oriented, even if the resulting program isn't. [Mad88]
- A program can be Object Oriented, even if the language it is written in isn't. [Mad88]
- An Object Oriented program can be written in almost any language, but a language can't be associated with object oriented-ness unless it promotes Object Oriented programs. [Str91]

Albeit, in this thesis, Object Oriented Programming does not mean using an Object Oriented Programming Language (OOPL), but means implementing an Object Oriented Design into a suitable programming language of choice, which should result in an OO program.

Object Oriented Programming Languages provide mechanisms to support the Object Oriented paradigm and programming style. Such an OOPL simplifies the job of implementing an OO design, but an OOPL is not a must. Implementing an OO design can be done in virtually any language. Such an language may be C [Som96], but that does not mean C is an OOPL: it just enables the developer to write OO programs. [Str91]. (Likewise, using an OOPL does not mean you are thinking within the OO paradigm).

Bjarne Strourstrup his opinion is of very high interest as Bjarne Strourstrup goes on saying that a language enabling you to write OO programs, should not be considered if it takes an exceptional amount of effort to do it [Str91]. We regard this opinion as rather naive, as a language is only a mere tool in the tool-suite: responsibilities of the tool-suite can be divided among the tools, as long the end result is an usable environment.

Indeed, in the article "The Silver bullet" [Jr.87] the author of the document is searching for the silver bullet that is needed in software engineering. He makes a difference in the essential and the accidental mistakes. The essential ones are inherent to the problem in software engineering: fashioning complex conceptual invisible constructs which are hard to understand and grasp. Accidental mistakes are related to "non-inherent problems" such as problems regarding programming mistakes, environments providing little comfort, etc. In his search for the silver bullet he asserts that nor yet another high level language nor tool-support are candidates for the silver. However, he takes more hope in Object Orientation, the paradigm, than in any other thing as a possible silver bullet: High level languages and tool-support both only solve accidental mistakes, not the essential ones.

2.1.4 Summary

Object Orientation is actually a name compromising different aspects, all relying on the concepts that have sum up earlier.

With the definition of OO Programming in mind (I.e. implementing an OO design into a language) it are thus those concepts that needs support in a development environment.

A development environment is a tool-suite that only helps the developer in removing the accidental features and thus providing a pleasant environment, independent in how the responsibilities are divided among the elements of such a tool-suite. Such an element is the programming language. This language may or may not be an OOPL: for OO an OOPL is not a must, and C may be a candidate target language for implementing an OO design.

2.2 Object Oriented Programming in C

2.2.1 Introduction

An OOPL knows the concepts of OO and provides for those mechanisms to ease the implementation of an OO design. Although C does not know the concepts of OO, Object Oriented Programming in C is possible. It is been said that C enables the technique of OO programming.[Str91]

This is possible because of two things. First, due its nature, C is quite a flexible programming language. Secondly, the constructs behind the mechanisms of an OOPL, are in fact quite often simple and straightforward constructs that can be applied in a language such as C.

Of course different approaches to such constructs are possible. Therefore the object model of C++ will be taken as a guideline. Although the implementation of such an object model may differ from compiler to compiler, most C++ object model implementations are grosso modo the same. Using the C++ object model it will be shown how OO programming in C can be done.

In fact, the first C++ compilers where mere C++ to C compilers (such as CFront [Lip96]) using the techniques explained in this section. Those techniques are also the basis for doing OO programming using Glib-C and this section wants to show how they work and how they are achieved. By doing so, an initial understanding of OO programming in C is acquired, such that the subject can be better understood.

Of course, the intention is not to explain the C++ object model in detail. An interesting and more complete look at the C++ object model can be found in "Inside the C++ Object Model", by Stanley B. Lippman. [Lip96]

2.2.2 A C++ Example

Explaining the mechanisms and constructs behind the object model of C++ will be achieved using a simple example. Consider the two C++ classes in figure 2.1. For this example some basic knowledge of C++ is required.

Class "Car" contains only one data member which represents the name of the car. The data member is private and can only be accessed using the corresponding getter and setter. There is also a virtual function member "printInfo". "Car()" and "∼Car()" are the constructor and destructor respectively.

```
class Car {
    public:
        Car();
        ~Car();
        void setName(char*);
        char* getName();
        virtual void printInfo();
   private:
        char *name;
};

class LuxeCar :  public Car {
    public:
        LuxeCar();
        ~LuxeCar();
        void setLuxetax(int);
        int getLuxetax();
                void printInfo();
   private:
        int luxetax;
};
```

Figure 2.1: Example C++ class "Car"

Class "LuxeCar" inherits from class "Car". It adds to class "Car" a data member representing the luxe tax of such a car, along with a getter and a setter for this data member. In class "LuxeCar" we assume the "printInfo" function member is overridden (actually extended), in such a way it both prints the basic data (the name of the car), and the data inherent to a luxe car (namely the luxe tax).

2.2.3 The basics: ADT style programming

Behind the scenes C++ uses an Abstract Data Type (ADT) approach to model objects and function members. In accordance to [Bud97] an ADT is realized by

1. having a type definition (think of the class)
2. a set of operations to manipulate the object (encapsulation)
3. making data in the object only available through those operations (encapsulation and data hiding)
4. and being able to create multiple instances of an ADT (i.e. an instance conforms to an object).

It is clear that those properties satisfy the base properties of OO programming. ADT programming may be regarded as a subset of OO programming and is often called Object Based (OB) programming.

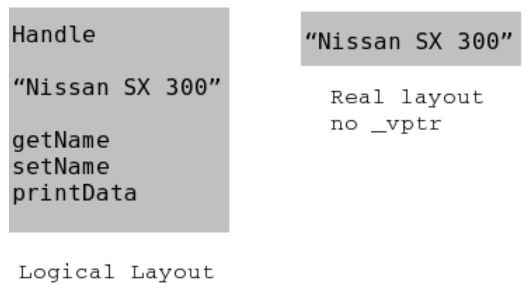


Figure 2.2: Possible memory layout for an instance of class ”Car”

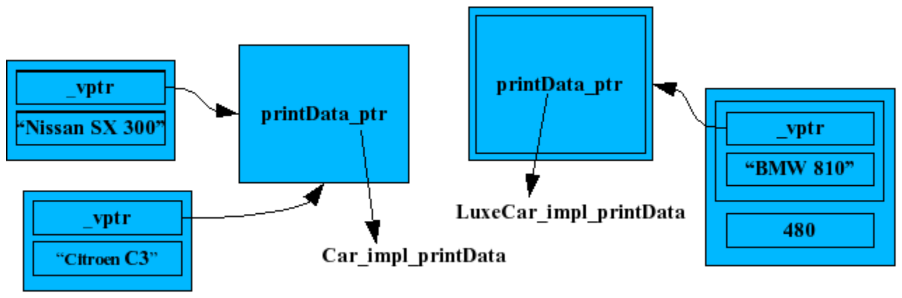


Figure 2.3: Objects of the same type share the same class structure

This suggests that a first step to OO in C is thus using an ADT style programming. An ADT is basically an embodiment of the concept of encapsulation, which plays a central role in OO.

ADT style programing in C is possible and shall be discussed at the end of this section.

2.2.4 Objects and Memory Layout

An instance of class ”Car” is an object with type ”Car”. Conceptually the memory layout of such an instance would contain an identity, values for the data members (state) and its function members (the behavior). Therefore an instance of class ”Car” could logically look like as in figure 2.2.

But the state of an object is only determined by the values of its data members. Function members are static and do not have to be embedded in the object. But also the identity is omitted: the identity in C++ can be as well the start address of the object structure. In figure 2.2 a possible real memory layout is depicted: a simple ”structure” only containing the values of the data members of the object.

2.2.5 Function Members

Next, lets take a look at the function members. In C++ there is made a clear distinction between virtual function members and non virtual function members. The first are function members that can be *overridden*, i.e. giving an implementation depending on the type. E.g. class "LuxeCar" shall override "printInfo" in such a way it prints besides the name of the car, also the value of its luxe tax. It are the virtual function members that are responsible for polymorph-ism in C++. Non virtual function members are function members that cannot be overridden.

As might be guessed, non virtual function members are the simplest case. Let's take a look at the setter and getter in class "Car", i.e. the function members "setName" and "getName". In the next small piece of code we create a new instance of the class "Car" and set its name to "Nissan SX 300".

```
Car *car1 = new Car();
/* the next line is of interest */
car1->setName("Nissan SX 300");
```

The "setName" function call is said to be a message "setName" sent to the object "car1". Behind the scenes, this small C++ code snip-let gets translated into machine code. Pseudo C code is used to show what is going on behind the scenes.

```
Car_setName(car1, "Nissan SX 300");
```

C++ translates the message "setName" to object "car1" to a mangled function call with the "car1" object as first implicit parameter. Already, ADT style programming becomes visible. Objects in memory are mere structures containing the values of their data members, and non virtual function members operate on them using the object as the first implicit argument. [1] Indeed, as long classes without virtual function members are used, you are dealing actually with ADT style programming covered with some syntactical sugar. The interesting part in this story are the virtual function members.

Virtual function members in C++ object model are often no more or no less than function pointers, i.e. pointers to a function. Such a pointer gets assigned a different function depending on its dynamic type. E.g. if the "printData" virtual function members gets overridden in class "LuxeCar" the function pointer gets the function assigned that is defined in "LuxeCar".

[1]Indeed, introspecting an C++ object file symbol table with an utility like "nm" on UNIX/linux systems shows the mangled ADT style functions.

Those function pointers are often saved in a so called Virtual Function Table, often referred to as VFT or vtable. The VFT is compromised by the *class structure*. Every object has a pointer to this class structure. This means, per type, there is only one class structure in memory, but possibly many object structures. All those objects of the same type have a pointer to this class structure. This is visualized in Figure 2.3. The pointer to the class structure is indicated by "_vptr".

When the "printData" virtual function member is called, the corresponding function is called from the VFT. This may look like the next code snip-let in pseudo C, assuming that slot 2 points to the function implementing the virtual function "printData".

```
/* The virtual function member call printData in C++ */
car1->printData();

/* gets translated by the compiler to : */
(*(Car*)car1->_vptr[2]) ((Car*)car1);
```

2.2.6 Inheritance

Using inheritance we extend a class. It is also possible to *override* virtual functions members. First lets take a look at extending the class itself. A class inherits all the attributes and function members. This means that the corresponding object structures and class structures in memory extend the existing ones with the data members and extra virtual function table slots respectively.

Another way put, the object structure of the subclass has the object structure of the superclass at the beginning of its memory layout. The same goes for the class structure of the subclass. Multiple inheritance is achieved the same way, except there are multiple object structure instead of one. In practice, multiple inheritance complicates the object model drastically, therefore multiple inheritance is not very useful (i.e. feasible) in C.

The so called "_vptr" in the object structure of the superclass can be reused. If an virtual function gets overridden, the function pointer in the extended class structure gets assigned the overridden function member. Calling virtual function members has already been explained.

2.2.7 Shared data members

Shared data members or class-wide variables are in C++ known as static data members. Of course these static data members are stored in the class structure instead of the object structure, along side the VFT.

Often a class has associated so called static function members. A possible view or implementation of this concept are function members operating on the class structure rather than the object structure. In C++ (and Java) however it is allowed to use static function members on objects[2]. But since an object has a pointer to the class structure, this can be easily translated.

2.2.8 Putting it all together

The constructs behind the concepts in C++ are very simple and show the way to OO programming in C. As can been seen, the whole syntactical sugar seems rather thin. Of course, there is more about the object system than what is shown up till now.

Example given, when a certain class gets instantiated for the first time both the object structure and the class structure are instantiated. The second time, only an object structure is created, and the pointer to the class structure needs to be updated. There must be a type system that does some administrative work and keeps track of, among other things, the class structures. That being said, let's put all those constructs above together and see how we could do OOP in C in practice.

In "The basics: ADT style programming" there is already touched how the cornerstone of OO can be achieved in C. Encapsulation is achieved through two things. First, Requiring that all data is put in C structures and that all uses of those go through functions. Second, defining a type (a C structure) whose data is hidden and protected from the user thanks to the concept of a file in C[Mey97]: declaration is put in the header file, and implementations are put in the source file. The same goes for the "function members".

In fact, to define a class in C, two structures have to be created: the corresponding object structure (the ADT) and the class structure. The first member of the the object structure and the class structure is the object structure and the class structure of the the parent class respectively. The object structure contains the data members of the class. The class structure

[2]Invoking static function members on object structures is however strongly discouraged in the C++ and Java world

contains the shared variables of the class and the VFT, which in turn is simply a bunch of function pointers.

For practical reasons a common base class should possibly be made, like in Java or C#. Such a base-class in C would basically be very simple. The object structure contains a pointer to the class structure (the "_vptr") and the class structure may contain some data members to store runtime type information (RTTI). In C++ e.g. the first entry of the VFT actually points to a structure containing RTTI.

When using pointers to object and class structures, inheritance (and thus also polymorph-ism) does work. E.g. passing a pointer to an object structure of LuxeCar to function Car_setName does work. The object structure does not get sliced so no information is lost (since only the pointer is passed, not the values of the structure). In reality, the pointer to the object structure should be casted to avoid compiler warnings.
In C++ pointers (and references, which are some sort of pointers too) are indeed visibly used to achieve polymorph-ism.

Chapter 3

Glib-C

3.1 What is Glib-C

3.1.1 Introduction

The C library ”Glib” is an open-source cross-platform general-purpose utility library, which provides common data structures, abstractions, utilities and functions. In this respect, ”Glib” can be compared to the standard library of C++. ”Glib” also offers a type and object system for C which allows the C developer to implement an OO design with ”ease”. The latter is of course of high interest.

The ”Glib” library originated from the GIMP project, the GNU Image Manipulation Program, a project started by two Berkeley students called Spencer Kimball and Peter Mattis. When this project moved from the GUI toolkit ”Motif” to its own toolkit ”GTK” (GIMP ToolKit), the ”Glib” library was born. However, at that time ”Glib” did not contain a type and object system. Later-on, ”GTK” became ”GTK+”. ”GTK” used a flat hierarchy of widgets, ”GTK+” added an inheritance hierarchy. Therefore a type and object system was introduced in GTK+, known as ”GtkObject”. When GTK+ reached version 2.0, the type and object system moved out of the toolkit into the Glib library. There it grew into an advanced independent system, that could be used in any C project.

Today, ”Glib” and its type and object system are used in a variety of projects, including GIMP, GNOME, Evolution and Gstreamer. All those projects prove that OO programming with Glib is certainly possible and are a legacy of the power of Glib.

We refer to C code written in an OO fashion using Glib and its type and object system to "Glib-C". "Glib-C" is not official terminology and shall be defined exactly later on. However, the given description should suffice for now. Note however, that "Glib-C" is plain standard C and does not add syntax or the need of a precompiler.

3.1.2 Type System and Object Model

The type system which includes the object model, resides in the GType module. GType is the GLib runtime type identification and management system and provides among other things a class based object model. Upon this object model a common base class is made, which is called "GObject". All Glib-C classes should inherit from this common base class. I.e. GType provides the type and class system and GObject is a common root class, comparable to "java.lang.Object" in Java.

When talking about Glib-C its type and object system, we shall often refer to it as GType/GObject.

The Glib-C object model can be compared to that of C++: figure 2.3 is perfectly applicable to Glib-C. However, the Glib-C object model is far more runtime oriented than that of C++, and adds advanced runtime information and introspection. E.g. an object can be asked for its type name, its parent, which properties it has, ...

Another difference is that GType/GObject only features single inheritance, but adds support for interfaces.

An overview of the Glib-C object model:

- class based
- single inheritance
- support for interfaces, where classes can implement multiple interfaces
- virtual and non virtual function members
- reference counting mechanism for memory management
- advanced runtime type information
- query-able generic properties interface
- query-able signal system

The property system of GObject allows to name properties and enables to query for them. Using this system, properties can be set or get by name using a generic system. This feature also originated from the GUI world. (E.g. for GUI builders). E.g.

```
/* setting properties by name */
g_object_set(G_OBJECT(aCar), "mileage", 56, NULL);
g_object_set(G_OBJECT(aCar), "price", readPrice, NULL);
```

The signal system is introduced in the GObject common base class and is not related to UNIX signals. This signal system serves as a general purpose notification mechanism and allows for e.g. event driven programming: objects can emit events and corresponding handlers are invoked. Since GType/GObject originated from a GUI library, this system was introduced for event handling in a GUI application. Example given: the user could create a button in its application and could connect a handler to e.g. the signal/event "clicked". In this handler a corresponding action could be defined.

The use of signals in GUI applications is only one example. The signal system can be used in a wide variety of situations.

Signals are also used to name virtual functions. Those are so called action signals. User code can emit signals by name freely on objects, thus invoking corresponding virtual functions. This may be especially useful in combination with the fact that signals are query-able.

Generally, they are a mechanism for communicating between objects on a type safe manner, but loosely coupled between components, making it easy to create re-usable software components. In the next example a signal of a "car" object is connected to a handler.

```
/* connecting a signal handler (function member cb_car_drive_forward
 * of an object of class "Demo::App") to the signal "driven::forward"
 * from object "aCar" */
g_signal_connect(G_OBJECT(aCar), "driven::forward",
                 demo_app_cb_car_driven_forward, NULL);
```

3.1.3 Why Glib-C

Why use Glib-C rather than an OOPL such as C++? The prime reason the GTK+/Glib developers choose C instead of a language as C++ was exactly because they do not wanted to use C++.

At that time C++ compilers were not as reliable as today. Even still today, C is far more portable than C++. However, the situation surrounding C++ has drastically improved since then and that particular reason has lost some of its power.

However, it is feasible to do OO programming in C, so why not doing so? Besides that, the C++ object model misses some powerful RTTI mechanisms and advanced runtime-introspection (as it does not support access to any type information on runtime [IL90] [1]). Those mechanisms are very useful in the area of e.g. GUI applications. The type system and object model in Glib-C provide some powerful features not found in C++. The first is far more runtime oriented. Above that, Glib-C provides a powerful query-able signal system and query-able property system, both also usable in e.g. GUI applications. Remember that runtime introspection sometimes is seen as an important factor for a modern OO environment (although we don't regard it as an essential concept in the OO paradigm. See Chapter 2).

Exactly because of those reasons Trolltech Qt and the open source desktop environment KDE[2] have extended standard C++ with a more runtime oriented object model. In that system a common base class QObject is introduced and therefore it is often referred to as QObject/C++. QObject/C++ uses a preprocessor to compile the code into standard C++ code and to generate type information that can be queried for. This preprocessor is called called MOC (Meta Object Compiler). Therefore this extended C++ is also often referred to as Moc/C++.

Glib-C is also designed to be easily integrated into existing systems. I.e. it should be easy to bind or wrap Glib-C based libraries into other languages. Because it is written in C, it is accessible from a lot of languages like C++, Objective-C, Python, Java, .Net and many others. In fact language bindings generators for Python, Java and .Net do exist. Also runtime introspection plays an important role in this area. E.g. very dynamic (typed) languages like Python, use this feature to successfully bind Glib-C libraries. Also the signal system that uses closures to marshal arguments of the signals, is designed to communicate through system and language boundaries.

The memory management model system was also designed to be easily integrated in existing systems that are using garbage collecting. This is possible because of the reference counting mechanism part of Glib-C. Also, the

[1]Only via the use of dynamic_cast the type can merely checked

[2]Qt is a cross platform GUI toolkit made by Trolltech. The desktop environment and developer platform KDE is built upon this toolkit

-	object model	language complexity	ease of use
C++	very static, no root object	great deal of syntax added compared to C	good knowledge of C++ required
Obj-C	dynamic object model, root object (Object or NSObject)	only 5 constructs added compared to C	generally easy
MOC C++	static object model with improved RTTI, meta-information, root class (QObject), signals	very much the same as C++	very much the same as C++
Glib-C	static object model. Advanced RTTI, meta-information, root class (GObject), signals	standard C	a lot of LOC compared to others. No syntax for OO

Table 3.1: Comparison between C++, Objective-C, MOC/C++ and Glib-C

destruction phase of an Glib-C based object is split in 2 phases: dispose and finalize.

3.1.4 Comparison

For a better understanding of what Glib-C is, a comparison between Glib-C, C++, Objective-C and Moc/C++ is drawn up. Those languages have been chosen because they are all related to "C with OO extensions". All languages are also compiled to native byte code as target language, that in contrast with e.g. Java and .Net.

C++ is a well known OO/hybrid programming language by Bjarne Stroustrup and is widely used. Objective-C is the less known OO "variant" of C and is mostly used in the OpenStep/Cocoa frameworks such as in Mac OS X. As already been indicated, MOC/C++ is a C++ variant that gets precompiled by MOC into plain C++ code. The latter is used in Qt and KDE.

Table 3.1 summarizes the differences between the 4 platform. C++ is the first entry because it is the defacto "C with OO" language. Compared to the 3 others, C++ has a very static and simple object model. This allows the C++ compiler to do the bulk of the work and minimize runtime overhead. However, RTTI information in C++ object model is very minimal.

Objective-C has a very dynamic and runtime oriented object model. It features e.g. late object binding (all the others do not). Objective-C is very different from C++. It is inspired by Smalltalk-80 and it only adds 5 constructs compared to C. It remains thus a very small and compact language. Next, Objective-C is not only a superset of C qua syntax but also retains the semantics of C. C++ poses restrictions that prohibit some standard C code to be compiled with a C++ compiler.

Objective-C also has a root class, called Object (plain Objective-C) or NSObject (in case of OpenStep, Cocoa). Such a common base class provides a more consistent and easier to comprehend object model and diminishes the need for templates. Using a common base class is often used to achieve genericity that is in C++ normally achieved by templates. E.g. in Java and Glib-C the common base class is indeed used for genericity. This is a design choice and has its supporters and opponents. Even Trolltech, creators of MOC/C++ defend MOC by saying, among other things, it takes away the need for templates; a feature that does not work well on every C++ compiler.

Of course, this is not the prime reason for the existence of MOC. The MOC precompiler generates meta-information for the C++ class and translates the slightly modified C++ code into standard C++ code. MOC enables this way advanced RTTI and query-able information. As Glib-C, it features an generic query-able property system as-well a query-able signal system. Because of the precompiler used, this requires a lot less of LOC compared to Glib-C, where everything has to be done manually by the developer or a separate tool.

3.1.5 A short example

A very short example shows how Glib-C code looks like. In the next example we assume a class hierarchy with base-class "Demo::Car". An array of cars (whether they are Cars, LuxeCars or something else) is iterated and every car is printed on an XML buffer. Depending on the runtime type of the object, the "writo_to_xml" virtual function member will write different data on the "xmlBuffer" object.

The variable "aCar" is a GObject based object. Using ADT style programming we invoke the function members, such as "demo_car_write_to_xml". This function member "write_to_xml" is actually virtual, i.e. behaves polymorphic-ally depending on the dynamic type of the variable "aCar". (E.g. it can be of type Demo::Car, Demo::LuxeCar, Demo::Limousine, ...).

The GPtrArray and GString data structures are part of the Glib library, but are not Glib-C classes, but pure ADT's.

```
/* ... */

GPtrArray *carList = g_ptr_array_sized_new(nrCarsRead);

/* A database of cars is read and put in the carList datastructure */
/* ... */

GString *xmlBuffer = g_string_new();
DemoCar *aCar = NULL;

for(i = 0; i < carList->len; i++) {
    aCar = (DemoCar*) g_ptr_array_index(carList, i);

    g_string_append_printf(xmlBuffer, "<!-- Car nr %d -->\n", i);
    demo_car_write_to_xml(aCar, xmlBuffer); /* virtual function call */
    g_string_append_c(xmlBuffer, '\n');
}
```

3.2 Glib-C Idioms

3.2.1 Naming Conventions and Idioms

OO programming in C is possible, but because C does not know the concepts of OO itself, we need to somehow enforce OO in C. In Glib-C this is twofold. First there is the technical side. Glib-C uses the GType type system that provides a class based object system. Upon this a common base class is created, GObject.

Second, because of missing syntax there is a need for a set of idioms. An idiom may be defined as "The specific grammatical, syntactic, and structural character of a given language" or "A speech form or an expression of a given language that is peculiar to itself grammatically or cannot be understood from the individual meanings of its elements". [3]

I.e. C has a certain syntax but we want a set of rules that give a particular meaning to some functions or other elements of the language. E.g. every Glib-C class must be accompanied with a type safe checking macro which has always the same form.

Languages as Java do have an "is-instance-of" alike operator to test whether a certain object is an instance of a certain class. I.e. using this operator the

[3]Source: The American Heritage Dictionary of the English Language, Fourth Edition

dynamic type of an object may be tested. C does not feature such an operator. In Glib-C however, a macro is created for every class and interface that has always the same convention. I.e. such an operator always must look like <PACKAGE>_IS_<CLASSNAME>. Thus in the case of a class "Demo::Car" there must be a "DEMO_IS_CAR(obj)" macro available.

Users expect those macros to be there, so they can be used. They also expect that the macro has always the same form, so they not have to search for the type checking macro, but readily know how it looks like. I.e. Glib-C can only be "spoken" if certain idioms are met.

As may be noticed from the previous example, Glib-C idioms boil down to a set of naming conventions to some elements (like macros) that are needed within Glib-C. Some things are required, some things are prohibited, and so on. It may be seen as a disadvantage of Glib-C that it requires such a strict set of naming conventions. On the other hand, requiring such strong naming conventions is a good thing. Thus actually, Glib-C (almost) enforces the developer to use a good coding style.

Coding conventions can indeed play an important role anyhow, also in e.g. Java which may be regarded as a clean OOPL. Why are code conventions that important? To answer this question we give an extract of the Java coding conventions from Sun:

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

All of those reasons are important. In Glib-C the third reason is especially important as will be noted later on.

3.2.2 Glib-C Definition

Glib-C is indeed more than just C code programmed in an OO fashion using Glib and its GType and GObject modules. A set of idioms must be respected before C code can be called Glib-C. Therefore, the need for an exact definition emerges.

We define Glib-C as next: Glib-C is only Glib-C if and only if three requirements are fulfilled. Those are, Glib-C is standard C code

- where the Glib library is used
- where OO fashioned programming is applied using the GType/GObject type and object system
- where the Glib-C Idioms are respected

C code using the GType and GObject modules of Glib that do not follow the Glib-C idioms, may not be regarded as Glib-C.

Those Glib-C idioms are defined in this thesis in Appendix A and are discussed in the next section, where an overview of those idioms is given.

The Glib-C idioms herein defined emerged from the already existing naming conventions that are used in todays GType/GObject related programming, such as in GTK+. We expanded and formalized those naming conventions further in such a way they are ”stronger”. However, this does not mean they are complete.

Indeed, the herein defined Glib-C idioms are ”strong”. They are designed in such a way that as much of possible information can be extracted from those idioms without the need for something else. Especially in the case in which only the header files are available, a common situation when using C libraries.

Nevertheless, this does not mean that weaker idioms could not be sufficient. Actually, throughout this thesis a weaker set of idioms will be defined. Nevertheless, rigid coding conventions are good and using the strong idioms are thus more then recommended.

In the definition of Glib-C it is not specified whether those idioms should be weak or strong. This suggest we may speak of Glib-C with strong or weak idioms. Nevertheless, the stronger the better.

In the next subsection an overview of those (strong) idioms is given.

3.2.3 Glib-C Idioms Overview

Glib-C knows a set of reasonably common OO concepts. A class in Glib-C has

- a class name

- a package name
- a type name
- a parent class
- data members and GObject properties
- virtual and non virtual function members
- GObject signals
- any number of implemented Glib-C interfaces

A Glib-C class must indeed reside in a package, hence has always a package name associated. E.g. a class with class name "Car" in package "Demo" is indicated by "Demo::Car". The concatenation of the class and package name is the type name, e.g. "DemoCar".

Two structures have to be created to declare a Glib-C class: An object structure, representing an instance of the Glib-C class, and a class structure, representing the virtual table and so on. The object structure must be named the type name of the Glib-C class, and the class structure the type name appended with "Class". E.g. "DemoCar" and "DemoCarClass" for the object and class structure respectively for a class "Demo::Car".

The function members of the Glib-C class are associated with the class through their names, as is common in ADT style programming in C. A function member name contains a reference to the package and the class name. Function members should be in lower-case. The first parameter of such a function is a pointer to an object structure of the Glib-C class. E.g. "demo_car_drive(DemoCar *drive, guint distance);". The name of the function member is actually "drive". "demo_car" is the prefix of the function member. However, when talking about the name of the function member often the complete name (i.e. "demo_car_drive") is used.

A Glib-C class is always accompanied with a so called Get-type function. This function has the signature "GType <prefix>_get_type();", e.g. "demo_car_get_type". An instantiate-ble Glib-C class (i.e. one that is not abstract) is accompanied with a "new operator", which can look like "DemoCar *demo_car_new();" or "DemoCar *demo_car_new_with_name(gchar *name);". (The last one is of course needed since C does not have a new operator such as in C++ and Java)

A Glib-C class is also accompanied with a series of convenience macros such as the one we already discussed: the type safe checking macro. There are also macros for type retrieval, type safe casting (both for object and class structure), class structure type checking and class access from an object.

Regarding Glib-C interfaces most of all these rules come back. Nevertheless, there are some differences. First off all Glib-C interfaces can not inherit from another. Secondly, an interface does not have data members. (Both can have an abstract meaning of properties though).

However, the conventions regarding the naming conventions of the function members remain. Interfaces are also accompanied with a set of convenience macros (which are of course different that those used with classes).

The biggest difference is that Glib-C interfaces only compromise one real structure containing the virtual function table and pointers to default signals handlers. The name of this interface structure must be the type name (e.g. ”DemoXmlSerializable” in case of ”Demo::XmlSerializable”) appended with ”IFace”. There is however a dummy structure with the type name, representing an instance of a class implementing the interface.

Chapter 4

Object Orientation Power of Glib-C

Although C is not an OOPL, developers can implement their OO design into C. I.e. C enables the developer to use an OO programming technique. This is possible using e.g. the Glib library and the usage of idiomatic programming. The end result is called Glib-C code.

Glib-C is indeed a possible choice if a developer wants to apply OO programming. However, an important question raises here: How fit is Glib-C as a viable alternative OO environment? We translate this question into "What is the OO power of Glib-C".

Stroustrup [Str91] asserts that C indeed may enable OO programming but should not be considered as a viable OO platform as it takes exceptional amount of effort. Also e.g. George Staikos asserts that OO programming in C is a lot less productive than in C++ because of the high overhead in Lines of Code (LOC).[1] However, both look at the language in pure isolation. It's rather naive and short-sighted to look at the language in such a way, as the language is a mere part of the whole tool-suite. To investigate the OO power of a platform, not one element of the tool-suite but the whole tool-suite must be considered and evaluated.

Just as a normal car is normally not driven on a simple sand way, program code written in any programming language should not "drive" on a simple notepad application. E.g. Smalltalk-80 would not be such a productive environment if only the language as a mere tool is considered. It is however the

[1]Actually, George Staikos was comparing GTK+ C code with QT C++ code. See chapter 5 for more information

whole environment with the language and all its technical aspects (pure OO, dynamic typed, runtime engine, ...) in combination with tool-support such as the refactoring browser, that make up the end experience to the developer.

To analyse the OO power of Glib-C in a broader perspective, this chapter will set up a few criteria for an healthy usable productive OO environment. Next it will describe some possible problems and how we will assess them. Basically, this chapter introduces the next chapters and will outline a run to a possible solution to the problems introduced in this chapter.

4.1 Criteria

To answer the question of the OO power of Glib-C, we use a small set of criteria. Those are:

- can the developer express his OO ideas and designs into Glib-C with ease and comfort?
- how productive or comfortable is the Glib-C environment? (e.g. automatic code completion)
- can modern software techniques surrounding the OO paradigm that allow the developer to grow software applied to Glib-C, those include:
  - prototyping and code generation
  - testing and testing framework
  - refactoring

The first item of these criteria is of course essential. Indeed, Glib-C allows us to implement an OO design. But should we do it if takes exceptional amount of effort? [Str91].

The second one relates to simple tool-support, in which we might think of the open source Java IDE Eclipse and its ease of use.

The last of these criteria is maybe more important than the others. Even if the developer is able to express his OO ideas with a certain level of ease, even if the developer has the joys of a standard IDE, how fit remains the OO platform to apply the paradigm of growing software? A paradigm that goes hand in hand with OO.

All those criteria eventually may deduce to the question: How well suited is Glib-C to build tool-support around it, eventually leading to a comfortable OO environment? We ask this question, keeping in mind that the language is only a mere tool among other tools. As we stated earlier, using a language as a standalone tool and looking at such a tool in total isolation, is rather naive. It's like judging a car on its engine alone. Indeed, it is a very important part of the car, but the chassis, safety systems, and so on are equally important.

How the responsibilities among the tools are distributed is not that important, as long the developer has a comfortable environment to its disposal. E.g. in a tool-suite based upon the Java Hotspot VM (and hence the Java language itself), the Ant build tool, JavaDoc, JUnit and the Eclipse IDE, the language (being a small and clean OOPL) may possess relatively enough responsibilities to provide a great deal of the comfort that is eventually provided by the whole suite.

In Glib-C however, it is clear that the language on its own will provide only a small deal of comfort and that the other tools in the suite have to take more responsibilities.

E.g. C++ is a hybrid OOPL that provides a lot of features and takes a lot of own responsibilities. However, a C++ compiler is in consequence a very complex piece of software and it took many years before complete and reliable C++ compilers became available. In Glib-C the responsibility is divided among the language (the compiler) and the Glib library, but provides in the end a comparable set of features.

It is clear that the success or the failure to integrate Glib-C into such a tool-suite is dependent on how successful information from Glib-C code can be deduced.

4.2 The problem

Although Glib-C is widely used in e.g. GNOME, there are some clear problems with Glib-C. These problems can be divided into 2 major categories. The first major problem is inherent to the fact that C does not know the concepts of OO and hence, does not provide any syntax for them. It is obvious that this makes information deduction quite difficult as there is no syntax to rely on.

The second one is partly a consequence of the first major problem. I.e. the Glib-C OO system is nude: the internals are visible and this has several consequences:

- Glib-C contains a lot of boiler plate code, i.e. always returning template code.
- the developer is distracted from the problem he is solving by focusing a part of its attention on the nude OO system
- Glib-C is not as easy to get started with as e.g. Java

Indeed, Glib-C contains a lot of template code. Examples are the convenience macros and the Get-type function of the Glib-C classes or interfaces. This template code increases the number of Lines Of Code (LOC) considerably compared to e.g. (Moc/)C++ or Java.

The second consequence is clear. In Java e.g. the developer is released from handling memory management manually. This enables the developer to focus his attention more to the problem. In Glib-C, the developer has also to deal with the internals of the type and object system.

As a consequence of this, Glib-C is not easy to get started with. First, people may have to understand how an OO system internally may work, next they have to understand how Glib-C works, and finally, since Glib-C lacks the syntax for OO classes, there is a lot less familiarity.

The last problem and its 3 consequences can be solved by a certain degree if tool-support is available. Having a programming language standalone is rather naive, but on the other hand tool-support must be possible. Thus the last problem boils more or less down to the first one.

However, being able to build tool-support for Glib-C and solving the 3 latter issues are inter-weaved, but also different. The latter 3 are clear problems for which clear goals can be set and that can easily be checked if they are in the end solved or not.

4.3 A run to the solution

Fortunately Glib-C knows the concepts that return in every well known OO environment. Examples of those or not limited to those we have used to compare Glib-C with (see Chapter 3). E.g. besides C++, the same concepts return in Java and .Net. Those concepts are (but not limited to):

- classes
- interfaces
- packages
- methods
- inheritance

Those concepts may return in those languages under different names or different forms. E.g. methods in C++ are called virtual function members and packages are known as namespaces. Also, (standard) C++ does not know interfaces, but often so called pure abstract base classes are used in C++ to provide signatures to classes. However, the names of the concepts are not important but the concepts itself are.

The fact that Glib-C knows those concepts, although C itself lacks any syntax for it, is very hopeful. Above this all, the fact that Glib-C is only valid Glib-C if the herein defined idioms are respected, takes a lot of hope that we are able to deduce the required information.

However, the fact we require such strict code conventions, which the idioms basically are, raises some critical questions that will not be ignored.

To investigate to OO power of Glib-C, prototypes have been created that should allow the developer to express his or her ideas into Glib-C with ease and should enable the developer to grow software with Glib-C. Those tool-support prototypes compromise the criteria used to indicate whether Glib-C is a viable object oriented alternative.

For this reason there are prototypes created regarding code generation, unit- and regression testing and refactoring: a Glib-C class/interface browser, a Glib-C class generator, a Glib-C method completion demo, a unit testing framework, a unit test stub generator, and an (incomplete) refactoring browser. In the next coming chapters the concepts and their relation with Glib-C will be discussed in great detail.

In the next 3 chapters, each of these elements of growing software are described along side with the tools we build for it. Although the focus is on being able to grow software in Glib-C with the help of tools, the issue of comfort is not forgotten. Each chapter will discuss what information was needed from the Glib-C based source code and libraries, how it was deduced (most of the time this means: "what idioms did we need? did the idioms suffice?") and what the possible problems are with how we deduced the required information.

In the end, all this information is collected and will be thoroughly discussed to eventually draw a conclusion on the OO power of Glib-C and the things we eventually needed to come to that point.

Most of the prototypes itself were written in Python, except for the unit testing framework GUF, which was written in Glib-C. We used the PyGTK and PyORBit bindings to GTK+ and ORBit[2]. Python was chosen because it an interactive, scripted, OO language which is perfect for fast prototyping. Using Python in combination with the GTK+ toolkit in which it is possible to describe the GUI interfaces in XML (which are drawn with the GTK+ GUI builder Glade), enabled us to make the tools in a very short time and with a great deal of flexibility.

[2]ORBit is a free open source implementation of CORBA used in GNOME, to enable communication between components

Chapter 5

Prototyping and Code Generation

5.1 Introduction

In this chapter we take a look at trying to build toolsupport for prototyping and code generation in Glib-C. For this we take a look at what information is needed from the Glib-C code, how it is retrieved (i.e. what idioms are important), and what the potential problems are. We will also use this scheme in the next two chapters about testing and refactoring.

However in this chapter we will, before doing so, also take a look at code generation itself, as it tackles a problem inherent to OO programming in C: as can be guessed, code generation may provide a solution to the overhead in lines of code.

5.1.1 Prototyping and Code Generation

A prototype is a software program developed to test, explore or validate a hypothesis. Prototyping is the act of creating such a prototype. We distinguish between a exploratory prototype and an evolutionary prototype.

An exploratory prototype is exactly what its name implies: it is a prototype to explore whether certain goals are possible. E.g. they are used to experiment with User Interfaces (UIs), to validate functional requirements or to test new technologies. Those prototypes are known as throw-away prototypes: after being created they serve no longer any purpose for the developer.

An evolutionary prototype is meant to evolve into the final product: from an initial design a first implementation is drawn up. While redesigning and refactoring the prototype along the way, the software product is grown. This is effectively a form of incremental and iterative software development. Indeed, the concept of an evolutionary prototype is very important in the paradigm of growing software.

The C language is by nature not very fit for exploratory prototypes. Nevertheless, this is sometimes a simple necessity. However, creating an initial program from an initial design is of course very feasible. In practice the developer may use code generators such as ”class wizards” to create code stubs from his or her design.

In Glib-C this is especially important as a lot of boiler-plate code or template code is generated for the developer. This reduces the work load drastically. Code generation from an initial design tackles thus parts of the two major problems described before: we are able to prototype and more comfort is provided to the developer.

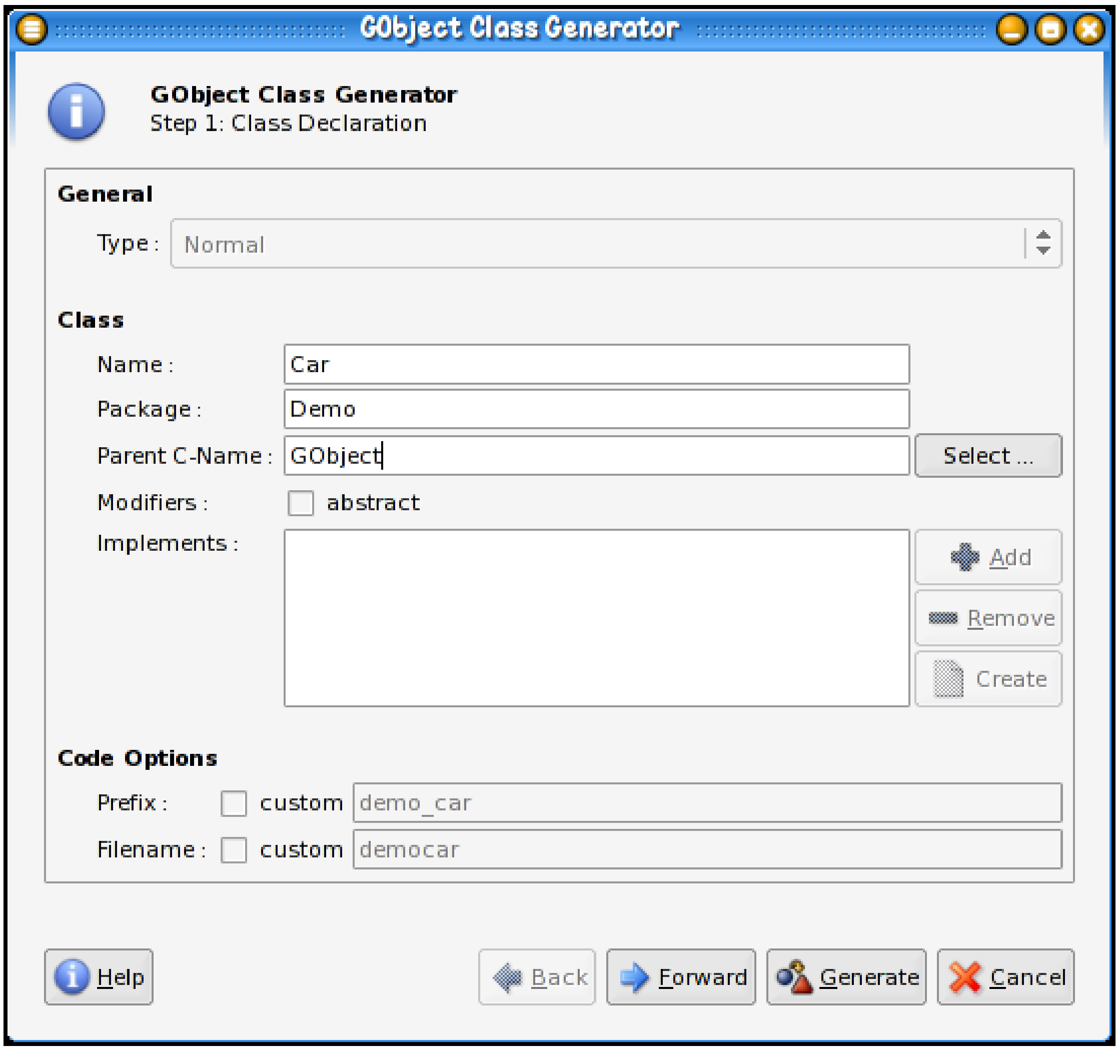
However, so called active code generation and completion are also considered. E.g. adding a function member to a Glib-C class, after the code has been generated and edited. The latter may already be considered a refactoring action. Regarding code completion, we think of presenting all the methods of a class while the developer wants to sent a message to an object of a certain type: what remains to do for the developer is to select the method he or she is searching for.

5.1.2 The Prototyped Tools

We created two relevant prototypes; a wizard-like Glib-C class generator and a method selection prototype.

The Glib-C class generator is loosely based upon the Java class wizard found in the Eclipse Java IDE. However, this class generator adds functionality to support Glib-C specific concepts like signals and GObject properties. Our Glib-C class generator is depicted in figure 5.1.

Basically the prototype allows the developer to fill in a few entries. The required entries are ”class name”, ”package name” and ”the type name of the parent”. The latter can also be selected from a class hierarchy maintained by the generator. From there on, code can be generated. However, the class



Class

Name : Car

Package : Demo

Parent C-Name : GObject

Modifiers : ☐ abstract

Implements :

Type : Normal

Prefix : ☐ custom demo_car

Filename : ☐ custom democar

Figure 5.1: A class generator for Glib-C

generator allows to specify data members, GObject properties, signals and virtual- and non virtual function members.

The method selection prototype is very simple: it maintains a class hierarchy from some Glib-C libraries. When trying to invoke a message to an object, the prototype presents all function members of the object. In Glib-C this is reasonably easy as we do not need to know the static type of the object: since we are dealing with functions that contain the name of the class (E.g. "demo_car_" for "Demo::Car"), the tool can immediately present all the function members associated with the class and its parents. I.e. the way we implemented this prototype is by typing the prefix of the function member, after which the prototype returns a list of all the function members of the class associated with that prefix.

More on these prototypes will be made clear during the rest of this chapter.

5.2 Code Generation

5.2.1 Class Stub Code Generation

Given a class "Test::EmptyObject" which extends a class "G::Object", we want to create an implementation of this class in C++ (or MOC/C++ for that matter) and Glib-C. This class is not further specified: nor data members nor function members are given. The resulting code should contain the ability to create and destroy the class.

I.e. the C++ code stub should consist of the class declaration, the empty constructor and the empty destructor. The Glib-C code stub should provide the Get-type function, the new operator function, the empty object, the instance and class initialization functions, the dispose and finalize function members (chaining up with their parent) and of course the convenience macros. The resulting code is depicted in appendix B.

The resulting code of both target languages differs quite a lot. The most important one is the number of lines of code (LOC). The C++ class consists of 20 LOC while the Glib-C class consists of 100 LOC. This is a considerable amount of code.

This issue has been covered by George Staikos "A Quick Cost Analysis of Qt vs GTK". He asserts that "the resulting code is typically 30%-60% smaller than the GTK equivalent" on porting a GTK+ application to Qt. Of course, this is comparing toolkits with different APIs and thus affecting LOC differently. However, the fact that GTK+ is Glib-C based, plays an important role in the huge difference between LOC of applications using both toolkits.

However, using a language as a standalone tool a quite naive approach. Remember the idea of the language being a mere tool in the tool-suite. Indeed, using code generators, creating a class in Glib-C or (MOC/)C++, requires filling in three fields and one click. I.e. the fields "class name", "package/namespace" and "parent class". Both environments require the same amount of work and time, and have the same impact on productivity. LOC is a bad metric anyway, but in this case LOC becomes very deceiving.

What is important is the fact that both C++ or Glib-C know the same concepts and that we can specify them independent of our target language.

5.2.2 Active Code Generation

Although a class generator solves an issue with plain Glib-C, generated code stubs are only the starting points. From there on software is grown. A more important issue is thus being able to evolve the product. E.g. adding functionality. This is where active code generation comes in. Lets consider adding a virtual function member to the "Test::EmptyObject" class.

Adding a virtual function member in C++ requires basically adding 2 "items" in the code: the virtual function member declaration (1 LOC) and the function member definition (2 LOC). So basically, adding a stub virtual function member in C++ requires an additional 3 LOC. This is depicted in figure 5.2.2.

In Glib-C we need to add 5 "items": a function pointer in the class structure (1 LOC), the wrapper function member in the declaration of the class (1 LOC), the wrapper function member definition in the source code (3 LOC), the implementation function of the virtual function member (2 LOC) and the initialization code (in the "class_init" function) that assigns the function pointer an implementation (1 LOC). In total, adding an empty virtual function member to a Glib-C class requires adding 8 LOC. This is depicted in figure 5.2.2.

Summarized, adding a stub virtual function member in Glib-C requires 266% of LOC compared to C++: this is a considerable amount. When starting from empty code stubs, we started with 20 and 100 LOC in C++ and Glib-C respectively. But adding functionality does not grow the code size linear: for every virtual function member added to the class the code size of

```

/*
 * Header
 * ------------------------------------------------------------
 */

namespace Demo {
    class Car {
        ...
        // >> +1 LOC : declaration of new virtual function member
        virtual void new_vfunc();
        ...
    };
};

/*
 * Source
 * ------------------------------------------------------------
 */

// >> +2 LOC : implementation of the virtual function member
void Demo::Car::new_vfunc() {
        /* -- an implementation here -- */
}
```

Figure 5.2: Adding a virtual function member in C++ requires an additional 3 Lines Of Code (LOC)

the Glib-C class grows harder than that of the C++ class. I.e. the gap of LOC between the two platform widens more and more.

Again, this shows the overhead in terms of LOC inherent to OO programming in C and a possible reason for not doing it. In the case of normal code generation the extra LOC is indeed very deceiving when LOC are regarded as a means to measure productivity. In here the issue is somewhat more complicated.

Tool-support can of course help in here: it can provide functionality to add a virtual function member. In Glib-C this certainly pays of: one entry and one click later the function member is added. In C++ this may be handy too, but is less essential, as writing 3 LOC is hardly any more work then filling in a dialog. However, C++ is a clear case in which the tool "the language" takes enough responsibility on its own, in such a way that other tools surrounding the language should take less responsibility. In the case of Glib-C it is the exact the reverse. This should not be a bad thing however: how responsibilities are distributed among the tools does not matter much, as long the end result of the tool-suite is a pleasurable experience.

```

/*
 * Header
 * -------------------------------------------------------------
 */

struct DemoCarClass {
    GObjectClass parent;
    ...
    /* >> + 1 LOC : function pointer in class structure */
    void (*new_vfunc) (DemoCar*);
}

...
/* >> +1 LOC : (wrapper) function member declaration */
void demo_car_new_vfunc(DemoCar *self);

/*
 * Source
 * --------------------------------------------------------------
 */

/* >> +2 LOC : implementation function of virtual function */
void demo_car_impl_new_vfunc(DemoCar *self) {
    /* -- an implementation here -- */
}

/* >> +3 LOC : the function member (wrapper) definition */
void demo_car_new_vfunc(DemoCar *self) {
    DEMO_CAR_GET_CLASS(self)->new_vfunc(self);
}

void demo_car_class_init(DemoCarClass *klass) {
    ...
    /* >> +1 LOC : assigning implementation to function pointer */
    klass->new_vfunc = demo_car_impl_new_vfunc;
}

```

Figure 5.3: Adding a virtual function member in Glib-C requires an additional 8 Lines Of Code (LOC)

However, while generating code starting from a small set of high level concepts like "class name", is very feasible and easy to do, there may be a problem with active code generation in this case. I.e. the tool must be able to fill in the extra code nicely (In chapter 8 we will take a look at adding a signal to existing code, covering the some problem).

Another example of active code generation is method completion: the tool-suite presents a list of function members related to an object. To be successful we need to deduce two things from the Glib-C code: the tool needs to know the class hierarchy (of the libraries and the project itself) and of course the static type of the object the developer wants to send a message too.

As function members are associated with their classes by means of their prefix, we choose to implement our prototypes in such a way we could exploit this feature. Because this omits the need for a code cross reference, our approach simplifies the job.

5.2.3 The Issue of Maintainability

Tool-support can help in generating code. This attacks two problems with Glib-C at the same time: the issue regarding overhead in LOC is solved and the developer has tool-support for what he really wants: prototyping.

However, the real issue with code is being able to grow it and being able to maintain it. What is important in the paradigm of growing software is important in the world of maintainability too. Actually, both are somewhat related to each other. Maintainable code is very fit for growing it.

Maintainable code depends on different factors in very different areas, e.g. on design or on the code itself. The last one depends on such things as readability, naming conventions and so on.

How Glib-C itself affects maintainability is of course in the area of the program code. Clearly the high LOC affects this. For one, the more code to read, the harder it is to understand. Secondly, changing code requires in Glib-C generally more work than C++. E.g. renaming an virtual function member requires in the case of C++ changes at 2 places in the code, and 5 places in the case of Glib-C. (Supposing this virtual function member was not used somewhere in the code).

The first problem is undeniable. However, Glib-C code always uses the same pattern, the same template code. This in combination with the Glib-C idioms which imposes some strict naming conventions, makes code quite recognizable and readable. This is were naming conventions are a very good thing. Nevertheless, navigating through the code may be more troublesome because of the template code obscuring the rest of the program code. Of course, huge C++ classes and huge Glib-C classes suffer from the same problem ("where is that setter?"), and tool-support can easily be constructed.

The second problem is actually the problem on the area of restructuring and refactoring. This problem will be tackled in chapter 8. One may think it is clear that renaming a virtual function member in Glib-C requires more work than in C++, and hence, the need for tool-support in the case of Glib-C is higher, compared to C++. But, the real problem in renaming virtual function members is not renaming the definition and declaration itself, but making sure that 1) it can be renamed en 2) that all uses of this virtual function member are adapted to the new name. Practically, renaming virtual function members, which is basically a refactoring operation, requires tool-support anyhow.

5.3 Information Deduction

To enable tools to provide support for prototyping and code generation certain information needs to be extracted from Glib-C code and libraries.

5.3.1 Class and Interface Hierarchies

In case of the code generation and method selection tools, the tools need to maintain a class hierarchy. This class hierarchy can be extracted from the source code given only the Glib-C header files. The Glib-C idioms require to have the object and class structures to be declared in the header file. Those structures should also conform to strict naming conventions, making them easy to identify.

What is required is the existence of 2 structures with names "_TypeName" and "_TypeNameClass". Given the existence of those 2 structures, the prototype can assume those 2 structures compromise a Glib-C class. I.e. it is very important in the Glib-C idiom that the class structure is named "_TypeName" appended with "Class" and not something else.

Next the tool looks at the first members of the class structure. If those data members have the types "TypeName" and "TypeNameClass" in the object and class structure respectively, the tool is confirmed the 2 structures compromise a Glib-C class, and learns at the same time what the parent class is for the newly found Glib-C class.

The Glib-C idioms also encourage that the first data members of the object and class structures are named "parent". This may be used as an extra heuristic, but is probably not very useful as it adds little certainty. Besides that, this naming convention is not a must, i.e. is not required.

After scanning the header files a class hierarchy can be constructed.

Constructing an interface hierarchy is more or less likewise. Of course, interfaces in Glib-C have a flat hierarchy. They may have so called pre-requisites, but interfaces cannot inherit from each-other.

To search for Glib-C interfaces the tool searches for a dummy structure named "_TypeName" and corresponding public structure named "_TypeNameIFace", whose first member is a "GTypeInterface". I.e. structures starting with first element GTypeInterface are most likely Glib-C interfaces. Nevertheless, the tools trust on the "IFace" suffix to determine whether this structure is such an interface or not.

5.3.2 Function Members

In case of both prototypes, the tools need to know the function members of the Glib-C classes. The class generator indeed needs this information too, because it needs to test if parent classes have the new function member already declared. In this case the tool may suggest to change the name, to create a code stub to override the existing implementation or to go with this name anyway. (This is possible in Glib-C but is of course highly discouraged and no real good tool should allow the last option).

Because of the naming conventions of the type name, the prefix for the function members can be easily deduced. I.e. for "DemoCar" the tool knows the prefix should be "demo_car" (if the Glib-C idiom is respected). Another example for type name may be "DemoXMLReader". For this type name the corresponding prefix for the function members should be "demo_xml_reader". The tool can confirm this by checking the prefix of the Get-type function.

However, the tools need also to know what interfaces a class implements. They also need to know which function members such an interface provides. In case of having both source and headers files available to investigate, this poses not very great difficulty. In the Get-type function of the Glib-C class the interfaces are registered using their type names. Because Glib-C interface structures need to be called "_TypeName" appended with "IFace",

```
guint nr;
GType *types = g_type_interfaces(demo_car_get_type(), &nr);
for(i = 0; i < nr; i++) {
     g_print("%s\n", g_type_name(types[i]));
}
g_print("\n");
g_free(types);
```

Figure 5.4: How to query a Glib-C class for implemented interfaces

corresponding Glib-C interfaces can be found from the interface hierarchy. This indicates again the importance of naming a structure correctly.

A problem arises when only the header files are available. This is the case with libraries written in C, that the developer uses in his project. Such a libraries may be e.g. Microsoft Windows' Dynamic Loadable Libraries (DLL files) or Unix/Linux Shared Objects (SO files). When the developer uses such libraries, the libraries are accompanied with header files. This is a must in C/C++ to enable the usage of those libraries. It is very seldom that source code itself is available.

Indeed, this is the first time we need to make a distinction between **project code** and **used libraries**. Project code is the code the developer is working at: the code he or she is growing. Used libraries is code that the developer uses in his project. E.g. the developer may be writing an IDE using the GTK+ toolkit library. The IDE code itself is the project code and the GTK+ toolkit library is the code which the developer uses in its IDE project. For that GTK+ library, only the header files are available.

Current GObject/GType code does not show which interfaces are implemented by a certain Glib-C class from the header file only. Luckily, Glib-C features so called runtime introspection. Figure 5.4 shows how introspecting a certain Glib-C class for interfaces can be done in Glib-C.

Our class prototype creates a small Glib-C program that introspects a certain type. This is possible because only the Get-type function of the Glib-C class needs to be known. This small programs gets compiled and executed. During execution this program provides the tool itself with the required information. In this case, which interfaces the class implements.

5.3.3 Static Object Type

In case of the method selection demo, the tool needs to know what the static type is of the object the developer wants to send a message to. Because Glib-C is based upon an ADT style of programming in which the function members of the classes compromise the name of class itself, the developer can type e.g. "demo_car_", press a button and all the function members corresponding with Glib-C class "DemoCar" and all its parents and implemented interfaces are shown. Of course, the more traditional approach, in which the developer starts with the object itself, is possible too. The difficulty lies of course into retrieving the function members which is discussed in the previous sub section.

However, the first way we proposed omits the reason for having a code cross reference that tells the tool of which static type the object is. Our less conventional approach may also be faster. In reality however, both implementations should be available. Nevertheless, this is a case in which Glib-C is easier to tool-support than e.g. Java.

The traditional way of doing things in Glib-C may look like in the next description. The developer types the name of the object, e.g. "car", and presses a button combination. After this, a list of function members is presented. The developer chooses one and the code gets actively generated: the object is surrounded with the correct function member, the object itself is correctly casted and the input cursors waits in the argument list if there is any.

5.4 The importance of Idioms

5.4.1 Overview

This is a short summary describing which specific parts of the Glib-C idioms are used and are important to successfully deduce relevant information from Glib-C source code, required by our prototypes.

To successfully deduce information from Glib-C source code, it is obvious that it is important to know which the Glib-C classes and interfaces are. The cornerstone to successfully deduce information from Glib-C code is thus the Glib-C idiom regarding the naming conventions for the object, class and interface structures. These Glib-C idioms enable the tools to find Glib-C interfaces, classes and the parents of the identified classes.

Next, information retrieval about the function members is possible, because the tool can deduce the prefix for the function members from the type name of the Glib-C class or interface.

Finally the importance of the Get-type function for the ability to introspect the type at runtime is clear. This is a necessity when dealing with used libraries. The Get-type function is again known from the type name, which is found through the names of the found structures.

5.4.2 Issues: a Critical Look

However, there are some problems. The Glib-C idioms are strict and when available they can be trusted. Nevertheless, the fact we rely on such strict idioms itself may be a problem. Therefore, let's take a look at some possible deviations and how this effects information deduction, the prototypes and maybe the idioms itself.

E.g. the Glib-C idiom requires that both object structure and class structure must be declared in the header file. However, to create a non-inheritable class developers choose often to declare and define the class structure in the source file itself. Lets assume the Glib-C idioms allows this. In this case, how does the tool know it is dealing with a Glib-C class?

Of course we need to approach this issue from two views: when we are dealing with project code or when we are dealing with used libraries.

In the case of project code the tool searches for a "TypeNameClass" structure provided a structure "TypeName" was found. A cross reference may be employed to achieve this. If the tool finds a corresponding Get-type function, this may serve as a conformation. I.e. this does not differ much from our prototypes. The difference is that both source and header are scanned instead of the header file only.

However, if we are dealing with used libraries, this is not applicable as the class structure declaration and definition is in the source file instead of the header file. Of course, a set of heuristics may be used.

Those heuristics could be based upon the existence of the convenience macros. Two macros are of up-most importance and the user expects them to be there: the type safe casting macro and the type checking macro. (When the developer creates an non inheritable class there is of course no interest in the class structure casting macro and the class retrieving macro). Those macros are part of how a developer uses Glib-C based libraries: without those the developer will feel handicapped.

Of course, those convenience macros correspondent to strict naming convention too. So, what if the convenience macros do not correspondent with those naming conventions? Possibly finding macros with "_TYPE" and "IS_" in their names, the tool can guess they are Glib-C convenience macros.

In reality those heuristics prove to be quite reliable. Nevertheless a better method would be welcome. Such a method may be the "Roll back to GObject" method. This method determines if a certain object structure is part of a Glib-C class by following its first data member. An object structure first data member is an object structure of the parent class. So, if the tool discovers that the first data member is also structure, it can track this first data member of our possible object structure. This structure will be investigated and if its first data member is also a structure, the tool will track this structure further. This goes on until the tool reaches the GObject object structure or when it runs into a death end. When the tool reaches the GObject object structure, it confirms that the initial object structure is indeed part of a Glib-C class. From there on, it can find more information about the newly found Glib-C class.

This method is very feasible because of 2 reasons. For one, we are sure we can track the whole hierarchy to the GObject class (provided we are indeed dealing with a Glib-C class). This is possible because the developer can only work with header files of the libraries, if they can be compiled into his code. To be a valid class declaration, the compiler needs to find the class declaration of the parent. This requires the inclusion of the header file (directly or indirectly) with the declaration of the parent Glib-C class. This is true for every Glib-C class, i.e. for every parent the tool encounters. However, we may need a complete cross reference of all those headers to use this method elegantly.

Secondly, the fact we traverse some kind of hierarchy already assumes we are on the right track. This method has also an extra advantage: While scanning the Glib-C class, some of the parents immediately may be validated as Glib-C classes by using the typical strong Glib-C idioms (and corresponding heuristics). I.e. based upon the availability of both object and class structures accompanied with their correct names. If such a Glib-C class is reached, we do not have to traverse the whole hierarchy back to the GObject (although we recommend it). The big plus about it, is that we learn to know some Glib-C classes that were previously unknown, by scanning one particular presumable Glib-C class.

However, there remains a problem if the object structure is declared in the header file but defined in the source file. Although we found out that this in reality is a very seldom case, it may happen.

But as might be guessed, the "Rollback to GObject" method remains very useful in the case of project code, where the strict Glib-C idioms are not obeyed. This method is in this case completely reliable.

However, on the quest of searching a reliable method for the case of "used libraries" we need something else. Therefore there is a final solution available which is based solely upon the Get-type functions.

A Glib-C based library may be accompanied by a file containing all the Get-type functions of relevant Glib-C classes and interfaces in that library. Using runtime introspection the types and class-hierarchies are found, whereas we always used scanning files before. Only finding the corresponding function members should rely on scanning the header-files now. Actually, this is how gtk-doc currently works. gtk-doc is a documentation system in the likes of JavaDoc and is used for Glib-C code, especially in GTK+. (The name gtk-doc is historical).

But instead of a file which lists the Get-type functions for all Glib-C classes and interfaces, tools can search in the headers for those Get-type functions. If it would happen that the Get-type function is not really a Get-type function, instantiating the type fails and the tool discovers that this function belongs not to a Glib-C class or interface.

E.g. our tool used the existence of the Get-type function to confirm the prefix for the function members. Although most existent GObject/GType based C code confirms to the Glib-C idioms that we defined, not all classes used the correct naming conventions. Examining the prefix of the Get-type function, the tool was in almost any case able to determine the function members.

Of course, this requires that the Get-type function ends with "_get_type()". This is thus actually the only remaining requirement. The Get-type functions should have the signature "GType <aprefix>_get_type". The prefix <aprefix> can be in theory almost anything.

5.4.3 Idioms reviewed

A first possible change to the Glib-C idioms is not longer requiring to have the declaration and definition of the class structure in the header file. However requiring that the name of the class structure is appended with "Class" remains essential. Also, having the definition of the object structure in the header file is not required as runtime introspection can be used. However, if we want to work without runtime introspection then the Glib-C idiom can advice to declare and define the object structure in the header file always using the next form:

```
struct _TypeName {
    ParentClass parent;

    TypeNamePriv *_priv;
};
```

Secondly, the prefix used for the function members may deviate from the standard definition, since the prefix can be looked up via the Get-type function. However, deviating from the standard definition should not be encouraged, although being consistent is of course the most important aspect. Nevertheless, not conforming to the standard prefix definition should raise a warning.

Next, the name of a structure denoting a Glib-C interface, may be different from "TypeNameIFace", since an interface can also be recognized as a structure whose first data member is of type GTypeInterface. E.g. some developers prefer to call this interface structure "TypeNameClass", as it was a normal class structure. Whether the found structure indeed is a Glib-C interface can be confirmed, if there is an accompanying dummy object structure whose name is the first substring of the name of the interface structure.

5.5 Summary

Code generation from a class seems very simple. However, even this requires some information about the already existing code base, such as the class hierarchies and basic information about each of these classes. The code generator starts from very simple common concepts that return in many modern OO environments, and from these concepts the code is generated.

Such a code generator solves some problems inherent to OO programming in C: boiler plate code, code normally behind the scenes of an OOPL, is generated for the developer. This relieves the developer from a lot of burden. The LOC of Glib-C is also very deceiving, as an important part of LOC in Glib-C consists of such boiler plate code, which is then the responsibility of the tool support and requires very little work or time to create.

The same goes more or less for active code generation, where the developer wants to add e.g. a virtual function member to the class later on. Using tool-support, a lot can be automated, reducing the effort considerably.

The Glib-C idioms defined in this thesis are designed within mind that tools should deduce as much as information as possible from the source code, and especially from the header file only. However it is very surprising how Glib-C classes e.g. can be found and identified with relative weak idioms.

In the case of "used libraries" the only thing that is really needed is the Get-type function. There are only 2 rules: first it must be present (in the header file) and it should confirm to "GType <aprefix>_get_type()". Using this function, the tool can discover if this function belongs to a Glib-C type (whether it is a Glib-C class or interface) and if so, ask for information such as the type name, parent class, implemented interfaces, signals and GObject properties. Function members can be discovered by the tool using the prefix of the Get-type function. This requires of course that the prefix is consistently used, which is clearly a must.

However, for "project code" runtime introspection is not applicable (because code should not be compile-able at any time to enable tool-support) and the idioms become more important. However, the Get-type function may play an important rule in this too. (Which will be made clearer in the next chapters).

Nevertheless the naming conventions regarding "TypeName" for the object structure and "TypeNameClass" remain very relevant. This learns the tool we are dealing with a Glib-C class. Also the "Rollback to GObject" method can be used, but is not in very high interest because the tool can see the complete code. At least if the class structure can be found, if not, "Rollback to GObject" remains useful. However, because this code evolves (this in contrast with used libraries), the class structure must indeed be known. E.g. for adding a virtual function member. If the class structure cannot be found although the "Rollback" method suggests a Glib-C class, tool-support can ask for the class structure and possible fix it.

The fact that in the case of "used libraries" information can be deduced using quite weak idioms (relative to the ones defined) is because of the ability of runtime introspection: what is important to learn from this, is how we deduce the information we need is not very important, as long the tools can deduce the common concepts.

From a high level view, we need some concepts that always return in different environments. In Java e.g. we also want to need a class and interface hierarchy when creating a class from a wizard. Because Java is a valid OO platform, tools can learn what the class and interface hierarchies are. In Glib-C we need the same information. The way this information is deduced is not very important, as long it is possible on a reliable way. In Glib-C we require a set of idioms (which are already much weaker than the original strong idioms) which are part of the concept of Glib-C itself.

Chapter 6

Unit- and Regression Testing

6.1 Introduction

Testing strategies such as unit- and regression testing are an important part of the idea of growing software. In this chapter it is investigated how fit Glib-C is to apply those typical testing strategies used in well-known OO environments such as Smalltalk and Java.

6.1.1 Testing

Software testing is a critical element of software quality assurance [Pre97], also in the paradigm of growing software: every time the system grows, it must be checked if no errors crept into the system. The latter is actually known as regression testing.

Each time the system grows (i.e. functionality, modules, ... are added or changed), changes may introduce problems with functions that previously worked flawlessly. The activity of regression testing must ensure that no new problems crept in the system. If this is not the case, regression testing should help locate the error(s) such that they can be fixed. Regression testing is the re-execution of some subset of tests that have already been conducted, this to ensure that changes have not propagated unintended side effects [Pre97]. This implies that a regression test-suite should be ran regularly, ideally after each "increment".

Regression testing should be automatic and deterministic, i.e. do not require user intervention. Regression tests should also answer whether the tests ran successfully or not. In the first case it should answer ”true” or ”ok”. In the latter case it should answer ”false” or ”not ok”, and specify where the problems occurred in the regression test suite.

Such a set of tests that are regularly re-run (and adapted if needed) could be a set of unit tests. Unit tests are small tests that test a unit. A unit in an OO context is typically a class. This strategy allows to find errors on small units very fast. These tests are based upon white- and/or black-box testing techniques. However in an OO environment they are most often black-box tests. These threat the unit as a black box where the implementations of the unit is not known. Black box tests focus on the functional requirements of the software [Pre97].

In the Extreme Programming (XP) paradigm it is encouraged to first write the unit test, then the unit (the class) itself. This obliges the developer to think twice about the interface of its unit (class). In practice the developer writes a code stub of its class (using e.g. a code generator as in the previous chapter) and creates a unit test stub along side. Next, the unit test stub is implemented, afterward the unit itself. A weaker form of this technique consists of creating the unit gradually and immediately filling in the test stub.

Today there are different unit testing frameworks available for a wide variety of (OO) environments. A unit testing framework is simply a library that helps the developer to write unit tests. Such a library mostly provides a TestCase and a TestSuite class. The first is a baseclass for the unit test itself, the latter is simply a suite of TestCases (or even other TestSuites). Examples of these unit testing frameworks are SUnit for Smalltalk, JUnit for Java, NUnit for .Net (C#, VB.Net, ...) and CppUnit for C++. Also, Python comes with its own unit testing framework. Actually, SUnit was the first unit testing framework and many others (all the others mentioned here) are inspired on this unit testing framework.

6.1.2 The prototyped tools

We created a simple unit testing framework called GUF: GObject Unit-testing Framework. Unlike the other prototypes we created, this is not a throw away prototype and is intended to grow into a usable system.

GUF was based upon JUnit and ”ctest”. ”ctest” is a very simple basic unit testing library for C that has no notion of OO. GUF is written in Glib-C and consists of 4 major classes: Guf::Test, Guf::TestCase, Guf::TestSuite and

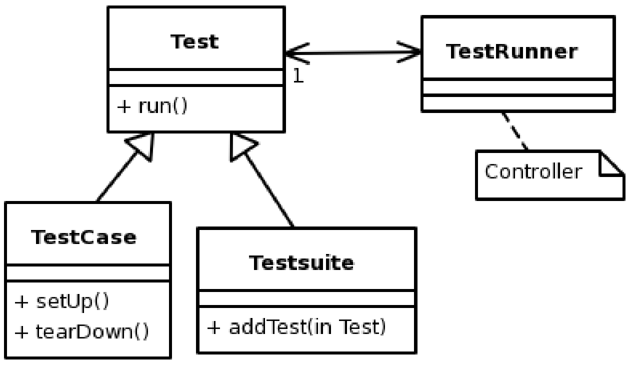


Figure 6.1: UML Diagram indicating basic design of GObject Unit-testing Framework (GUF)

Guf::TestRunner. Their relation is depicted in a simple class diagram (figure 6.1).

We created another small prototype: a small tool to create automatic unit test stubs for existing Glib-C classes. I.e. given a Glib-C class, this tool is able to generate a unit testing stub for that class, covering each of its public function members.

Also interesting is the ability to create unit testing stubs along side Glib-C class stubs, which should be possible with simple extensions to the code generator. However, this is trivial in the context of the previous chapter and is not discussed.

6.2 Information Deduction

GUF itself learns us very little about information deduction. The importance of GUF is of course that there is a unit testing framework for Glib-C, enabling us to speak about unit tests and testing in Glib-C. Although the simple ”ctest” library could suffice, GUF is more specialized featuring Glib-C signals and is modeled after traditional OO unit testing frameworks.

However, for creating automatically test stubs from existing Glib-C (stub) classes, information needs to be extracted. Such a tool proves to be more interesting. Again, the information the tool needs, is information about the class itself such as its name and the function members (also the one provided by implemented interfaces).

The intention of the tool is to generate a unit test from a given Glib-C class. This unit test itself is a class inherited from Guf::TestCase. Every function member of this unit test corresponds with a function member of the tested class. This enables the developer to test every function member in a clean environment. Of course, to generate such a unit test stub, all the public function members of the tested Glib-C class need to be known.

If the developer wants to create unit tests for existing library code without seeing the implementation, unit tests can be created to learn to understand the code [DDN03]. In this case the (strong) idioms and/or runtime introspection can be used to extract the required information.

However, software re-engineering is not our scope. More important is to create unit test stubs for and from classes in the developer's project code. In that case, the tool has to rely on a set of idioms only.

All of this has been thoroughly discussed in the previous chapter, and all what we have learned can be applied to this subject. However, there remain some interesting things that have not been discussed before. Those are related to abstract classes, extended or overridden virtual function members and specific Glib-C features such as GObject properties and signals.

6.2.1 Abstract Classes

Imagine that a developer wants to create a unit test stub for the class "Demo::Person". He or she right clicks on the name of the class in a class hierarchy viewer and wants to click the "generate unit test stub" item. However, the class "Demo::Person" is abstract and thus cannot be instantiated. Hence, creating a unit test stub for this class is impossible.

Determining whether a Glib-C class is abstract or not is again not possible from the header files only. However, the absence of the new operator may indicate (and should in the case of strong Glib-C idioms) that the class is abstract.

In the case of "used libraries" we again can rely on runtime introspection for certainty, which on its turn relies on simple naming conventions regarding the Get-type function. However, in the more interesting case of "project code", the Get-type function is again essential.

The Get-type function of the Glib-C class needs to register the class with the runtime system. In there, it is specified whether the class is instantiate-able or not (i.e. abstract). Since there are only a small number of options for registering a class, this can easily be tracked.

In the next code extract, the Get-type function of an abstract class "Demo::Person" is shown. The type ("Demo::Person") is registered using "g_type_register_static" and the last argument of this functions flags that this type is not instantiate-able.

```
GType
demo_person_get_type() {
    static GType this_type = 0;

     if (!this_type) {
         static const GTypeInfo this_info = {
             ... /* values for this_info */
             ...
         };
         this_type = g_type_register_static(G_TYPE_OBJECT ,
                                            "DemoPerson",
                                            &this_info,
                                            G_TYPE_FLAG_ABSTRACT);
    }
    return this_type;
}
```

6.2.2 Overridden/Extended Virtual Function Members

Another interesting case regards extended or overridden virtual function members. This is also interesting in the part of code generation too and has been shortly touched in the previous chapter.

Assume a Glib-C class which overrides or extends a virtual function member of a parent class. The unit test stub generator must create a unit test of this class. In this case it is of course important that the implementation of the subclass is tested in that unit test. Hence, the tool needs to know if the class extends or overrides a virtual function member of the parent class.

An alternative way is to test every function member that is inherited, but this makes unit tests very redundant and unnecessary big. Not to mention extremely boring to make.

In C++ or Java this is easy detectable, because the virtual function member is again declared in the sub-class. In C# the keyword ”override” is even being used. In Glib-C however there is no indication in the class declaration (i.e the header-file) that a certain function is overridden or extended. This is because the function pointer in the class structure gets assigned a new implementation in the source code, but this is not visible from the header.

A first important step is to recognize virtual function members. This however can be done from the header file only, and may be useful for a Glib-C type browser too. A function member can be recognized as virtual if the name of the function member stripped from the prefix, equals the name of a function pointer in the class structure. E.g.

```
struct _DemoCarClass {
    ...
    void (*print_data) (DemoCar*);
};

inline void demo_car_print_data(DemoCar *self);
```

Nevertheless, when tools need to know when this virtual function member gets overridden or extended in a subclass, they must have the complete source available.

What happens precisely if a virtual function member is overridden? In Glib-C this involves 2 steps. First creating the implementation of the overridden/extended function member, next assigning this implementation to the function pointer in the class structure. These 2 steps can be found in the source, giving us the required information. The next code snip-lets illustrates extending a virtual function member ”print_data” declared in ”Demo::LuxeCar”.

```
void demo_luxe_car_impl_print_data(DemoLuxeCar *self) {
    /* call parent implementation */
    DEMO_CAR_CLASS(parent_class)->print_data(DEMO_CAR(self));
    /* additional own code, extending behavior of the parent
     * class implementation
     */
    ...
}

void demo_luxe_car_class_init(LuxeCarClass *klass) {
    ...
    DEMO_CAR_CLASS(klass)->print_data = luxe_car_impl_print_data;
    ...
}
```

Using this information, i.e. when a function pointer of a super class gets assigned a new "implementation function", the tools know that the corresponding virtual function member is overridden or extended. In the case of the example and our test stub generator, the tool knows that it also must test "demo_car_print_data".

6.2.3 GObject Properties and Signals

A unit testing creation tool can also create stubs to test specific GObject features such as signals and properties. Since signals and properties need to be registered in the class initialization function, the tool can discover which properties and signals a Glib-C class has.

The class initialization function has the form "<prefix>_class_init". Registering signals and properties can only be done in a limited number of ways in the class initialization function. From these registrations a lot of information can be extracted. In the case of GObject properties the name and the type can be discovered from the registration function. In the case of signals, the name and the signature of the corresponding handlers can be determined.

In the next example a GObject property gets installed on the class "Demo::Car". This property "name" represents the name of a car and is both readable and writable. The name of the property (which is "name") is always the first argument of the g_param_spec_* function (such as g_param_spec_string, g_param_spec_ulong, and so forth). For unit testing, only the name and the type needs to be known, and this can easily be deduced from the source code.

```
g_object_class_install_property(
     gobject_class,
     PROP_NAME,
     g_param_spec_string(
         "name",                     /* name */
         "Name",                     /* nick-name */
         "The name of the Car",      /* blurb (description) */
         "",
          G_PARAM_READABLE | G_PARAM_WRITABLE
     )
);
```

For signals, we need to know the name of the signal and the signature of the handler functions. The name of the signal is the first argument of "g_signal_new", and the signature of the corresponding handlers can also be determined. In the case of the next example, the handler must return void and has 1 argument, an unsigned long integer.

```
demo_car_signals[DRIVEN]= g_signal_new(
    "driven",
     DEMO_TYPE_CAR,
     G_SIGNAL_RUN_FIRST | G_SIGNAL_DETAILED,
     G_STRUCT_OFFSET(DemoCarClass, driven),
     NULL, NULL,
     g_cclosure_marshal_VOID__ULONG,
     G_TYPE_NONE,                       /* return argument : void */
     1,                                 /* one pararmeter */
     G_TYPE_ULONG                       /* unsigned long */
);
```

If only the header files are available (i.e. in the case of "used libraries"), runtime introspection can be used to extract the same information.

6.3 The Importance of Idioms

6.3.1 Overview

The tools only require basic class information. I.e. the class must be recognized along side all its public function members. This and the needed idioms have been thoroughly discussed in the previous chapter.

Tools also need to know whether the scanned type is instantiate-able or not. This can not be deduced from the header files only. In this case, runtime introspection is needed. However, in the more important and more relevant case of project code, the tool needs to recognize the Get-type function in order to deduce the required information. Again, this has been thoroughly discussed in chapter 5, stressing the importance of this function further.

Next, the tool requires information about which virtual methods of the base classes were overridden or extended. This information can only be extracted in the case of project code. For this, naming conventions for the function pointers in the class structure, and naming conventions for the wrapper function members (the functions that call the functions pointed by the function pointers in the class structure) are needed.

Finally, the tools need to identify the class initialization function to successfully determine which GObject properties and signals are installed. Since installing/registering properties and signals can only be done via a very limited number of ways, there are no further idioms required than those for the class initialization function.

6.3.2 A critical look

As in the previous chapter, we take a critical look at the idioms that give us the required information.

The first obvious possible problem is related to abstract classes. Abstract functions cannot be found from header files only, luckily runtime introspection provides a solution.

A more striking problem is related to deducing whether certain virtual function members are overridden or extended. This is not possible from header files only nor through runtime introspection. Luckily, creating unit tests for existing libraries is not our first intent. Still, this might be a problem asking for a solution through e.g. extended query-able meta-data.

Nevertheless, discovering virtual function members from header files only is possible. Discovering which of these are overridden or extended is also possible in the case of project code, which poses no problems when strict idioms are applied. These idioms basically say that for the name of a function pointer of the virtual function member, there must be a wrapper function member with the same name, but prefixed with the prefix. If this is not the case, the function pointer belongs to a signal. However, in case of action signals there may be a corresponding wrapper function. But in this case those signals may be handled as if they were virtual function members.

Imagine the case in which the class structure was declared and defined in the source file. In this situation the Glib-C class is not inheritable, and virtual function members would be useless anyhow. I.e. for discovering virtual function members this is not a problem.

necessity another problem is when the required idioms are not met. From the header files only it cannot be determined whether certain function members are virtual. Heuristics may be applied, but it would not yield a reliable result. In the case of project code it can be determined if a certain function member is meant to be virtual or not. However, this would be too complicated. I.e. the strong idioms required regarding virtual function members are thus a neccesetiy and cannot be weakened.

Regarding GObject properties and signals, we were able to deduce the required information in both cases of project code and used libraries, in the former using runtime introspection.

Since GObject properties and signals can only be registered using a limited number of ways, detecting which properties and signals get installed is relatively easy. The registration of properties and signals happens in the class initialization function which has the form "<prefix>_class_init".

However, the class initialization function must be registered in the Get-type function. Using this information, the class initialization function can be found even if it is not conform to the strong idioms. Again, this highlights the importance of the Get-type function even further.

6.3.3 Idioms Reviewed

As in the previous chapter, the cornerstone for information deduction in Glib-C is of course the ability to recognize and identify Glib-C classes (and interfaces). The importance of the Get-type function was clearly highlighted. The Get-type function should be conform to the idiom "GType <prefix>_get_type()"

In this chapter we saw the further importance of the Get-type function, as it allows us both by reading this function, as with runtime introspection to determine whether a type is instantiate-able or not.

Using this function the tools can also determine the class intitialization function, through which the GObject signals and properties can easily be identified. Thus, the strong idiom for the class initialization function is not really needed (although of course still recommended for readability and to make it easier for the tools). For merely identifying GObject properties and signals after the class initialization function is found, no further idioms are required.

What still remains important are the names of the function pointers versus the names of the function member wrappers of the virtual function members: the name of this function pointer is the name of the function member wrapper without its prefix. E.g. for a virtual function member ”Demo::Car::drive”, the function pointer's name is ”drive” and that of the function member wrapper is ”demo_car_drive”.

6.4 Summary

Of course, tests can be written for Glib-C code, as it is possible for each piece of code. However, Glib-C may be only fit for such frameworks if creation of test suites can be automated. In the case of project code, complete test suites can be generated, even incorporating Glib-C specific features as properties and signals, from a fairly weak set of idioms.

The required idioms and used techniques have been thoroughly discussed in the previous chapter, however some additional idioms and techniques were required in the context of creating test suites. Again, the Get-type function plays a central role in this story.

In the case of ”used libraries” the tools are however not able to determine which virtual function members of the super classes of the tested Glib-C class, are overridden or extended in that specific class. This results in a less complete test coverage as we would expect. Luckily, ”used libraries” are not the first target of unit tests. Tests are mostly applied to the developers in-progress project code. I.e. the code that is being ”grown”.

Chapter 7

Refactoring

7.1 Introduction

In this chapter we take a look at refactoring in Glib-C: again, we ask whether it is possible to apply the technique described in this chapter. And if it is possible, how? For this we also take a look at refactoring in C in general, as there are some specific problems with refactoring in this language. We also take a look at refactoring in other platforms. This will learn us how to deal with some specific Glib-C problems.

7.1.1 Refactoring

Refactoring plays a central role in the idea of growing software. Refactoring is an act in which a software system is changed is such a way that is does not alter the external behavior of the code, yet improves its internal design [Fow00]. [JR97] defines refactoring as a behavior preserving source-to-source program transformation.

In the paradigm of growing software a product grows iterative and incrementally by small steps. Before adding functionality, the developer first redesigns and refactors the code. By redesigning and refactoring the code, the internal design is improved, such that the system anticipates further changes or additions better.

While refactoring, the developer should have a regression test suite at its disposal. Regression tests are an important aspect as they verify that the external behavior of the program is indeed not altered. Ideally, the regression test suite should be run after each refactoring.

Refactorings may be applied by hand. However, refactoring is a slow and errorprone activity. Tool-support and automated refactorings should help the developer, and make the act of refactoring an integral part of the software development cycle, as is required in the paradigm of growing software.

Refactoring gained wide-spread attention in the world of OO. However, nothing in the given definitions confine refactoring to OO environments only, as the act of refactoring is interesting and useful in non-OO environments too. There it is often referred to as restructuring, although the term refactoring seems generally well adapted.

Opdyke [Opd92] was the first to mention refactoring as it is known today. He identified and defined 4 low-level refactorings:

- creating a program entity
- deleting a program entity
- changing a program entity
- moving a member variable

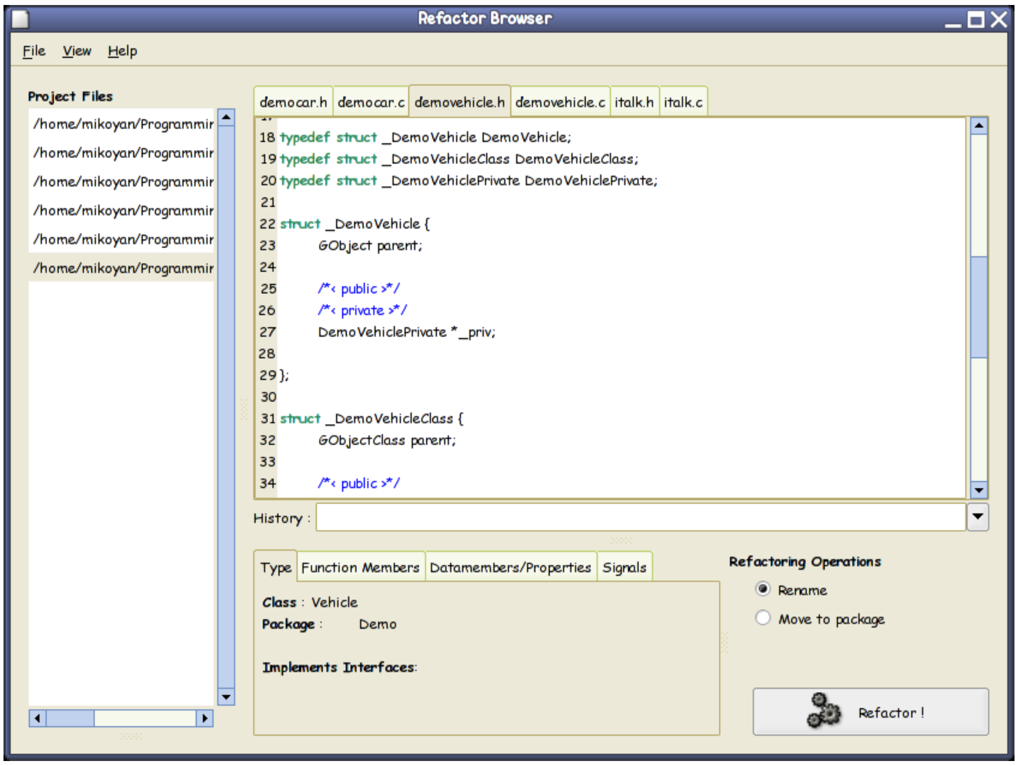
Where an entity is a class, a data member or a method (a function member).

In chapter 5 we discussed adding a function member in context of active code generation, which is an example of refactoring. Also removing a function member and renaming a function member, are examples of refactorings. The first research on restructuring in OO was conducted by [Ber91] and [E.91], and was focused on moving entities up and down in the class hierarchy. Fowler proposed an extensive yet incomplete catalog of OO refactorings. There he speaks of ”pull down data member”, ”push up data member”, ”pull down method”, and so on [Fow00].

7.1.2 The Prototyped Tools

In the context of refactoring we created the beginnings of a refactor browser inspired tool for Glib-C, i.e. a tool that allows to browse the project code, and that allows to apply refactorings on that project. The original Refactoring Browser is a tool implemented in VisualWorks and VisualAge for the Smalltalk language at the University of Illinois at Urbana-Champaign ([RBJ97],[Rob99] [1]). The Refactor Browser is the most successful refactoring tool up to date and serves as an inspiration for many new projects.

[1]http://st-www.cs.uiuc.edu/users/brant/Refactory/RefactoringBrowser.html


Refactor Browser
File View Help
Project Files
democar.h democar.c demovehicle.h demovehicle.c italk.h italk.c
History :
Type Function Members Datamembers/Properties Signals
Refactoring Operations
Refactor !

/home/mikoyan/Programmir
/home/mikoyan/Programmir
/home/mikoyan/Programmir
/home/mikoyan/Programmir
/home/mikoyan/Programmir
/home/mikoyan/Programmir

```
typedef struct _DemoVehicle DemoVehicle;
typedef struct _DemoVehicleClass DemoVehicleClass;
typedef struct _DemoVehiclePrivate DemoVehiclePrivate;

struct _DemoVehicle {
        GObject parent;

        /*< public >*/
        /*< private >*/
        DemoVehiclePrivate *_priv;

};

struct _DemoVehicleClass {
        GObjectClass parent;

        /*< public >*/
```

Class : Vehicle
Package : Demo

Implements Interfaces:

[✓ Rename]
Move to package

Figure 7.1: A Glib-C Refactor Browser

This tool was logically the most difficult one to build and gave us insights in the complexity of refactoring itself and specific problems related to refactoring in the context of Glib-C and C in general. Our tool however is a very fast prototyped program, and nor design or internal functionality, is based upon the Refactor Browser. A screen-shot of this tool is depicted in figure 7.1.

Basically, the refactor browser consists of a standard source code view for every project file. Every file is associated with a Glib-C type (a class or an interface). E.g. the files ”democar.h” and ”democar.c” are associated with the class ”Demo::Car”. The tool thus assumes that every header/source duo declares and defines one and only one Glib-C type. The tool displays information from the type associated with the file the developer is viewing. This information includes name, package, function members, data members/properties and signals. The developer can select an item (such as a function member) and apply refactorings to it.

This tool needs a cross reference of the source code (see section 7.2). For this we relied on the "Xref" command line tool from the Xrefactory project. [2] Xrefactory is a plug-in for Emacs and JEdit and is written in Elisp. Xrefactory provides a refactory framework for Java and C. It also contains the utility "Xref" which is able to create an extensive cross reference of the source code, there where the well known "ctags"[3] utility fell short.

7.1.3 Refactoring and C

Refactoring emerged from the world of OO. Nevertheless, nothing should confine refactoring to OO platforms only. Refactoring is about code evolution, about improving structure, about improving readability, about improving traceability. This are issues that are important for every developer, not only the OO developer. Therefore refactoring in C is as useful as in any other platform.

Example given, a C developer may want to in-line a function, extract a function, rename a parameter, rename a local variable, rename a function, rename a macro, move a function to another file, and so on. Sadly, most of the research of refactoring is focused on OO languages such as Smalltalk and Java. C refactoring tool-support is hard to find, and C seems to be almost neglected.

This may be very surprising since C is a very wide spread and widely adopted language and there are a lot of legacy systems written in C. However, as been mentioned, OO has been mainly the focus of refactoring. Secondly, refactoring requires changing small bits to the code at a time. I.e. making small steps. This requires fast edit-compile-run cycles and C is clearly not very fit for this, this in contract with Java and certainly with Smalltalk. Smalltalk may be seen as a platform that delivers very knead-able code, while C delivers code made from an harder substance: still knead-able but requires more power and time.

This of course raises the question: as a consequence of this, is C fit for growing software? In principle it is. We are able to prototype, test and refactor in C. The only issue is speed of the refactorings. But we omit the answer of this question for now, as there is another problem with C which we regard as the main issue with refactoring in C.

[2]http://www.xref-tech.com/xrefactory/main.html

[3]ctags is a common UNIX tool to generate tag files for source code and is used by e.g. the VI editor

```
/* an include directive */
#include <glib-object.h>

/* macros definitions */
#define DEMO_TYPE_CAR   demo_car_get_type()
#define DEMO_CAR(obj)   \
    (G_TYPE_CHECK_INSTANCE_CAST((obj), DEMO_TYPE_CAR, DemoCar))

/* conditional compilation */
#if PLATFORM_WIN32
    ...
    /* some win32 specific code */
#else
    ...
#endif
```

Figure 7.2: Examples of uses for the CPP

There is indeed another problem with the programming language C, as C adds a certain complexity. The programming language features a preprocessor that provides a macro language. The preprocessor includes "include" directives, "macros" and "conditional compilation" directives. The preprocessor of C is often referred to as CPP. Examples of those are given in figure 7.1.3

A compiler calls the preprocessor which replaces the CPP directives with the actual code, resulting in plain C code. The resulting code is then compiled to the target language.

"Analysis tools for C generally ignore the preprocessor. They apply their analysis to the output of the preprocessor, which, in the case of refactoring tools, is inappropriate. Programmers expect the results of a refactoring tool to still contain preprocessor directives. Therefore, refactoring tools cannot ignore the preprocessor. However, preprocessor directives are hard to handle for two main reasons: it is difficult to carry information of directives from the source code to abstract program representations and it is difficult to guarantee correctness in the transformations [GJ03]."

Luckily, there are solution for this problem as it is the primary research of the CRefactory research project [4]. Also the Xrefactory tool provides solutions to the most problems inherent to the CPP, but CRefactory want to provide a faster, better and more elegant solution.

[4]http://jerry.cs.uiuc.edu/ garrido/CRefactory.html

Interestingly, not only C suffers from these problems. Also the languages (MOC/)C++ and Objective-C feature the same preprocessor. Also the .NET language C# features a pre-processor inspired on the CPP. Although pre-processor of C# is limited to simple macros (nothing more than aliases) and conditional compilation, all of the solutions found for the CPP can be applied to C#.

7.2 Refactoring and Information Deduction

In the previous chapters we clearly stated which elements (idioms) are needed from the source code to deduce the required information. In this chapter we focus on the refactorings itself, and indicate per refactoring which information needs to be extracted from the source code, and how this was achieved.

Fowler proposed a comprehensive (yet incomplete) catalog of OO refactorings. However, we take a look at a very small set of refactorings that learned us a lot from a practical point of view. These are

- rename function member
- rename data member
- rename class
- move class to package
- push up function member
- extract function member

Finally, we take a look at very Glib-C specific refactorings like

- rename GObject property
- rename GObject signal
- add GObject signal
- remove GObject Signal

It should be clear that refactoring in Glib-C is basically refactoring in C. A Glib-C refactoring leads basically to a composition of ”standard” C refactorings and is driven by the Glib-C idioms. E.g. to rename a class in Glib-C, we know that at least the object and class structures need to be renamed. These objects need to be found (relying on the idioms), then a "standard" C refactoring can be applied to each of these objects. Thus in the case of renaming a Glib-C class, we have at least a composition of renaming two standard C structures.

Of course, we can only talk about standard C refactorings provided they exist. The CRefactory project has defined a catalog of C refactorings. Provided that there is clean solution available for problems related to the CPP, we can rely on these set of refactorings in standard C, to discuss refactorings in Glib-C.

7.2.1 Rename Function Member

The tool displays a list of known function members. The developer chooses from this list the function member he or she wishes to rename. Next the developer applies the rename refactoring. Deducing a list of function members has been thoroughly discussed before (see chapter 5).

Generally to apply a refactoring, there must be a few preconditions met [Opd92]. In this thesis we will not discuss a complete list of preconditions for each refactoring, as this would lead us too far in the context of this thesis. We will however discuss the most obvious ones and show what information is needed from the Glib-C code to handle these pre-conditions.

In the case of renaming a function member, it is clear that there should not be an existing function member with the same "new name". In Glib-C the tool can check for a function with "<prefix>_new_name". However, since the Glib-C idiom does not allow the same function member name in the same hierarchy, this idea can not be applied. Thus the tool needs to know the function member names of the parent too, which is again already been discussed. Note that refactorings are applied on code that is saint and compile-able.

If this pre-condition is met, the refactoring can be applied. Generally, the tool renames any declaration and definition of the function member, along side its usages. For this a good cross reference of the code is needed. A cross reference basically is a list of elements that are found in the code. Those elements are e.g. functions, structures, global variables, macros, data members from structures, and so on. The cross reference generated by the "Xref" tool from "Xrefactory" contains such a list: the declaration, definition, usages and so on of each element in the source code are located using the filenames, line- and column-numbers.

```
/* an implementation is assigned to the function pointer */
void demo_car_class_init(DemoCarClass *klass) {
        ...
        klass->drive = demo_car_impl_drive;
        ...
}

/* the function member wrapper calls the virtual function */
void demo_car_drive(DemoCar *self, guint distance) {
        DEMO_CAR_GET_CLASS(self)->drive(self, distance);
}

/* a subclass calls the parent implementation of drive */
void demo_luxe_car_impl_drive(DemoCar *self, guint distance) {
        DEMO_CAR_CLASS(parent_class)->drive(self, distance);
        /* specific Demo::LuxeCar code here */
}
```

Figure 7.3: Some cases in which the function pointer of a virtual function member is used

Using such information tools can safely rename a function member. Without such a cross reference, the tool could rename "too much". E.g. when renaming a global variable "foo", the tool can rename a data member "foo" in a structure "bar" too, if a global search and replace is used. With such a code cross reference, we are sure that only the right things get renamed.

If the function member to be renamed is a virtual function member, then also the corresponding function pointer in the class structure needs to be renamed, along side all its usages. Those usages are normally restricted to a few cases (see figure 7.2.1. First there is the assignment of an implementation to this function pointer. This happens in the class initialization function of the Glib-C class. Of course, this may happen in the class initialization functions of all its children too. Secondly, the function pointed by the function pointer gets invoked in the function member wrapper. Finally the function pointed by the function pointer may be called in the implementations of the children (i.e. subclasses call the parent implementation).

If the class structure can be identified and the function pointer has the right naming conventions, renaming the function pointer and its usages poses not any problem. Actually, if the tool discovers the function member is virtual, those conditions are already fulfilled.

A problem that may arise, is finding and renaming the function that contains the implementation of the virtual function members. With the strong Glib-C idioms this function looks like "<prefix>_impl_<name>".

Renaming a function member of an interface has likewise effects. The interface structure needs to be identified such that the function pointer can be renamed along side its usages. Also the implementation functions provided by classes which implement this interface should be adapted to the new name.

7.2.2 Rename Data Member

Renaming a data member is reasonably easy. Data members are part of the object structure or the private data structure and thus can easily found. In case of class-wide data members, they are part of the class structure. Using the cross reference provided by "Xref", the definition and usages of these data members are easily found, so they can be renamed (see "Push Up Function Member" for more information about this).

In the known list of function members, corresponding optional getters and setters can be identified and renamed. Normally, a setter has the form <prefix>_set_<name>, such as e.g. "demo_luxe_car_set_luxe_tax" for a private data member "luxeTax" in class "Demo::LuxeCar". In the case of setters, the parameter is normally the name of the data member. In this case, this parameter can also be renamed.

A problem arises when the data member is associated with a GObject property. As been mentioned before, a GObject property needs to be registered in the class initialization function. In the previous chapter we discussed how using this registration can be used to identify the GObject properties of a type. Now we also have to deal with these properties: there are some new elements to be discussed.

First of all, association between a GObject property and a data member is set by the name of the data member and the name of the GObject property. Secondly, registering a GObject property also involves an enum value. Normally this enum value has the form "PROP_<NAME>" and thus can be easily found and renamed.

It is clear, that the name of the GObject property and the name of the enum value should be adapted. However there is a deeper problem: GObject properties complicate refactoring severely as they are very dynamic entities. E.g. consider the next code snippet:

```
/* Function member Foo::Bar::do_something accepts
 * a G::Object and sets a property on it
 */
void foo_bar_do_something(FooBar *self, GObject *obj) {
    ...
    g_object_set(obj, "property1", "a string value", NULL);
...
}
```

Consider a class "Foo::Baz" with a GObject property "property1". Consider another class "Foo::Boz" that also has a GObject property named "property1". However, this class "Foo::Boz" is completely unrelated to "Foo::Baz". The developer wants to rename data member/property "property1" on "Foo::Baz" to "aproperty". The problem is that the tool does not know the static type of "obj", and hence does not know if "property1" needs to renamed in the "GObject::set" call. Basically, this means that the tool can not deduce the required information from the source code.

7.2.3 Rename Class and Move Class to Package

Renaming a class is an expensive action as it requires renaming various elements. First the object structure, the class structure and optionally the private data structure need to be adapted to the new name, along side the type definitions of those structures (the "typedefs").

Of course, the prefix of all the function members is affected. Thus all the function members need to be renamed, along side the initialization functions, the Get-type function, the new operator function, ... Also the class name in the convenience macros is affected, resulting in renaming all those convenience macros and their usages.

Finally, the name that is used in the registration in the Get-type function needs to be adapted to the new name.

Move Class to Package is in Glib-C the same action as renaming a class, as basically the type name of the class is affected. Remember the type name being the concatenation of the package name and the class name. I.e. we need to change all the elements that are changed in the Rename Class refactoring.

7.2.4 Push Up Function Member

To be able to push up a function member there must be made sure that all access to data members in the function member are restricted to the class the

```
1  #define DEMO_PERSON(obj) \
2  (G_TYPE_CHECK_INSTANCE_CAST((obj), DEMO_TYPE_PERSON, DemoPerson))
3  #define DEMO_CAR(obj) \
4  (G_TYPE_CHECK_INSTANCE_CAST((obj), DEMO_TYPE_CAR, DemoCar))
5
6  typedef struct _DemoPerson DemoPerson;
7  struct _DemoPerson {
8          GObject parent;
9          gchar *name;
10 };
11
12 typedef struct _DemoCar DemoCar;
13 struct _DemoCar {
14         GObject parent;
15         gchar *name;
16 };
17
18 DemoCar *car;
19 DemoPerson *person;
20 GObject *obj1, *obj2;
21
22 g_print("%s\n", car->name);
23 g_print("%s\n", person->name);
24 g_print("%s\n", ((DemoCar*)obj)->name);
25 g_print("%s\n", ((DemoPerson*)obj)->name);
26 g_print("%s\n", DEMO_CAR(obj1)->name);
27 g_print("%s\n", DEMO_PERSON(obj2)->name);
```

Figure 7.4: The ”Xref” utility is perfectly able to determine the usages of e.g. the data member ”name” of structure ”_DemoCar”, even in the case when C style or Glib-C style casts are used

function member is moved. The same goes for function calls (e.g. getters/setters).

Function calls can be easily checked using their prefix. For the data members we have to completely rely on the cross-reference of the code. The ”Xref” utility was perfectly able to determine the usages of these data members. Also in the case when typical C style casts or Glib-C type safe casts were used on the object structures.

E.g., consider the code extracts depicted in figure 7.2.4. There are 2 classes ”Demo::Person” and ”Demo::Car”. When trying to rename the ”name” data member of ”Demo::Person”, ”Xref” was able to indicate the correct usages of this data member. I.e. on lines 23, 25 and 27.

7.2.5 Extract Function Member

Extracting a function member involves selecting a part of a function and extracting it from this function member. From a C point of view, a function member is a mere function. Extracting a function is defined by [Gar] in her catalog of C refactorings. I.e. extracting a function member in Glib-C is extracting a function in C, no more no less.

To only difference is that the extracted function member must result into a Glib-C function member. I.e. has the correct prefix and has the first parameter ”self” in its signature. The extracted function normally takes only parameters for the variables that are used in that part of the function. However, the ”self” parameter must always be carried over to the extracted function, even if it is not used in that part of the function member.

7.2.6 Rename GObject property and Rename GObject signal

Renaming a GObject property is already discussed in ”Rename Data Member”. Through its dynamic nature we don't know if property names in calls like ”g_object_set” need to be changed or not. However, also signals are very dynamic of nature and suffer from the same problem. Some examples of signal usage:

```
/* a signal */
g_signal_connect(G_OBJECT(obj), "signal1", cb_handler, NULL);

/* the notify signal notifies listeners if a property is
 * changed. This is thus related to GObject properties */
g_signal_connect(G_OBJECT(obj), "notify:prop1", cb_notify, NULL);
```

Nevertheless, as with GObject properties, renaming the signal itself (but not its usages) is again feasible.

As a GObject property, a signal needs to be registered in the class initialization function. As we know, this registration requires a string representation of the name of the signal, a enum value and a function pointer in the class structure. It is clear that these 3 have to be renamed. The enum value should conform to ”<NAME>” and the name of the function pointer should be the name of the signal. The name of the signal can be found as the first argument of ”g_signal_new” (cfr. chapter 6).

A Glib-C class can assign a default object handler to the function pointer. This function should covey to the same ”impl” naming conventions as for virtual function members.

7.2.7 Add/Remove GObject Signal

Adding a signal (which also may be regarded as active code generation) poses no real problems.

However, as with adding a (virtual) function member, the code must be nicely integrated with the already existing code. For one, we have to add an enum to an enum type that is associated with the signals. This enumeration can be found as it does exist of capitalized signal names, and the name of the last enum value should be "LAST_SIGNAL". E.g. for the class "Demo::Car" for we which know it has a signal "driven", the next enumeration clearly is associated with signals:

```
enum {
    DRIVEN,
    LAST_SIGNAL
};
```

Secondly, a function pointer has to be added in the class structure. Finally, the signal must be registered with the class (most likely using "g_signal_new"). Since, the tool knows were the class initialization function is, and can search for other "g_signal_new" calls in that function, this can be nicely added in the right place (not just only in the class initialization function, but logically grouped with other signal registrations).

Removing a GObject signal itself is easy because the tool discovered the signal and knows its different elements. However, because the tool can not deduce correctly all the usages, the tool does not know if it can safely remove this signal. It is clear that a precondition for the removal of an entity is, that this entity is nowhere used anymore.

7.3 The Importance of Idioms

7.3.1 Overview

As in the previous chapters, the cornerstone of being able to deduce information is finding the Glib-C class itself. The same requirements return, such as finding the function members, data members, properties, signals. These have been thoroughly discussed before. However, besides finding those elements, we now have to deal with them, requiring some more idioms.

E.g. the tools now have to deal with properties and signals, implementation functions, convenience macros and so on. As in the previous chapters, we take a critical look at the required idioms.

7.3.2 A Critical Look

Almost all the idioms that are important for the refactoring "renaming a function member" have been discussed in the 2 previous chapters. Nevertheless, this time also the implementation function of a virtual function member is important.

If this function conveys ideally to the strong idiom "<prefix>_impl_<name>", this function can be easily found. However, if this is not the case, the function still can be found by looking up what function is assigned to the function pointer of the class structure in the class initialization function. The tool can search and replace for <name> in the found function. Also default object handlers of signals may be assigned to corresponding function pointers, which goes exactly the same as for virtual function members.

Renaming a data member is pretty straightforward once these data members are identified. For this we need the object, class and optional private data structure. Issues with the object and class structures have been discussed before. However, the private data structure can only be found if it confirms to the "<TypeName>Priv[ate]" naming convention.

A data member is often associated with a getter and a setter. These can only be fool-proof found if they convey to the strong idioms. Otherwise, these corresponding setters and getters cannot be identified and correctly renamed.

The biggest complexity when renaming a data member arises if the data member is associated with a GObject property. For one, the tool must be able to associate a data member with a GObject property. This can only be accomplished if the name of the GObject property and the name of data member match. Secondly, the tools now have to deal with the corresponding enum value of the GObject property. The name of this enum value must be "PROP_<NAME>". However, if this is not the case, the enum name can easily be tracked down because the enum value is registered using the "g_object_class_install_property" function.

The same goes for GObject signals. Again, a signal is associated with an enum value. This enum value should convey to the "<NAME>" naming convention. Also, a function pointer in the class structure is associated with the signal. The name of this function pointer should be "<name>". However, both may deviate from this as the enum value and the function pointer are registered with "g_signal_new". This clearly also affect the "Add GObject signal" refactoring. However, the enum value is not a parameter of this function, but is the array index of the static signal array. E.g., a signal "driven" on class "Demo::Car":

```
demo_car_signals[DRIVEN] = g_signal_new("driven",  .... );
```

A remaining issue with GObject properties are the so called "nicks" (simply the nickname of a property) and "blurbs" (a description of the property) of the GObject properties. If the name of a property changes, it is possible that the nick and the description need be adapted too. Although ideally this is probably not needed. The tool can however flag this items to be inspected by the developer.

GObject properties and signals complicate refactoring severely as the uses of these data members and signals cannot be deduced in a lot of common cases. However, this is a consequence of the technical side of Glib-C and not the idioms. This issue shall be discussed in the next section.

Renaming a class or moving a class to another package, is in Glib-C technically the same and has the same consequences, because both refactorings essentially alter the type name of the Glib-C class.

For renaming a class or moving a class to another package, we must find and identify the object, class and private data structures and the function members. However, we now also have to identify the convenience macros and the global signal array. The global signal array should ideally look like "<prefix>_signals". However, this array could be identified by looking at what the results of "g_signal_new" in the class initialization function are assigned. The tool can look for <prefix> in this found variable, and adapt it to the new prefix. However, the tool should fix the name of the found variable in such a way it is conform with the idioms.

What about the convenience macros? We already have discussed how a deviating prefix in function members can be identified. However, if the prefix does not correspondent with the strong idioms, it is most likely that the strong idioms regarding the convenience macros are not respected either. The question is of course how they still can be identified and renamed correctly?

A possible approach is to search for macros in the header file of the corresponding Glib-C type, whose name has "_TYPE_" in the middle. This macro should indicate the type convenience macro, which is an essential convenience macro and is expected to be there. Also, such a macro must be in the header-file or it is useless. The prefix of this macro is the "deviated package name" and the suffix the "deviated class name". Also the type safe casting macro is quite essential. Using this macro, the deviated names can be confirmed.

Also, as a consequence of the idioms, "the rename class" and "move class to package" refactorings are quite expensive as a lot has to be renamed. This also has a serious consequence in recompiling the C project, as a lot of object files will be affected.

In C++ e.g. only the name of the class has to be renamed and all the references to it. This may lead to a smaller set of affected source files (i.e. is a less expensive refactoring than in Glib-C) but affects the same amount of object files as in Glib-C: Internally C++ uses ADT style naming conventions for the function members incorporating (nested) namespace(s) and class name. All of these symbols in the symbol table need to be renamed too.

7.3.3 Idioms Reviewed

It is clear that it is essential that the name of GObject property match with the name of corresponding data member, as there is no other safe way to associate the property with the data member.

On the other hand, we can allow to relax the naming conventions of the related GObject properties and signals enums. Also a relaxation regarding the naming conventions of the "implementation" functions for virtual function members and default signal handlers is allowed. In principle, the names of these functions may be arbitrary, even not using the Glib-C function member prefix. Also, the name of the function pointer of a default signal handler should strictly spoken not be the same as the name of the signal itself (as this function pointer is also registered).

With the naming conventions of macros there are some more complex issues. Nevertheless, we can allow the usage of other <PACKAGE> and <NAME> conventions in the macros. However, this is only possible if the type convenience macro is of the form <A>_TYPE_<B>. E.g. for a class "Demo::Car" it is allowed to use "DM_TYPE_CAR", but not

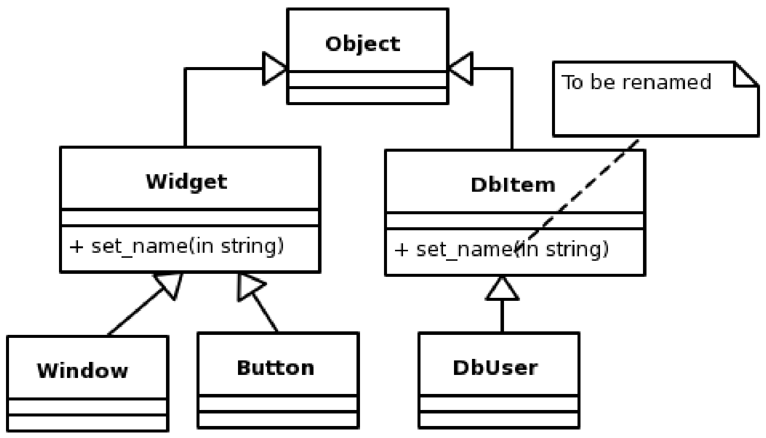


Figure 7.5: UML Diagram showing 2 different hierarchies in which the "base-classes" have a function member with the same name

"TYPE_DM_CAR" or "DEMO_CAR_TYPE". I.e. the order of the elements of the macro remain important.

7.4 Refactoring and other Platforms

In this section we take a look at other platforms and refactoring. This will learn us how to deal with similar problems as in Glib-C, i.e. dealing with very dynamic "late bounded" calls such as signals and those related to properties.

Currently the 2 most well-supported platforms for refactoring are Smalltalk-80 and Java. However, we also take a look at C++, Objective-C and Python.

The first tool written for automated refactoring was for the Smalltalk-80 environment [RBJ97],[Rob99]. However, Smalltalk-80 is not widely used outside the academic world and with the popularity of Java, refactoring tool-support for this language is arising. Currently, the Eclipse IDE provides a useful set of refactoring operations for Java.

7.4.1 Dynamic Refactoring

Smalltalk-80 is a dynamic typed language featuring real late binding. This means, unlike in C++ or Java, the tool cannot deduce whether a function member belongs to a certain static type. E.g. consider the class diagram depicted in 7.5. Two different class hierarchies features a function member "set_name". In the second class hierarchy, the developer wants to rename the function member ”set_name” in ”DbItem” to ”set_identifier”. However, the tool does not know which ”messages” with name ”set_name” belong to which class hierarchy since there is no static type to investigate.

In C++, Java and Glib-C for that matter, the required information can be deduced from the code allowing the rename the function member ”set_name”.

However, in Smalltalk-80 a wrapper method is used: the well-known Refactoring Browser uses method wrappers to collect runtime information. These wrappers are activated when the wrapped method is called and when it returns. As the program runs, the wrapper detects calls to the original method. Whenever a call to the old method is detected, the method wrapper suspends execution of the program, goes up the call stack to the sender and changes the source code to refer to the new, renamed method. Therefore, as the program is exercised, it converges towards a correctly refactored program. This type of refactoring is often referred to as dynamic refactoring.

The major drawback to this type of refactoring is that the refactoring is only as good as the available regression test suite. If there are parts of the code that are not covered in the regression test suite, than the refactoring will not be complete.

Also Objective-C and Python, which are also dynamic typed languages, are affected by the same problem. I.e. static typed languages are easier to refactor in this regard. Actually, renaming a function member in Glib-C is easier than in C++ or Java. In C++ or Java the static type of every object needs to be known, while in Glib-C this refactoring is completely driven by an idiom (i.e. by the prefix in the function members).

7.4.2 Solutions for Glib-C

However, Glib-C has some very dynamic features too: GObject properties and signals. The problem has been outlined before and some ideas incorporated in dynamic refactoring may provide a solution for this problem.

Let’s example consider ”g_signal_connect”. This function connects a signal of an object with a signal handler.

```
g_signal_connect(G_OBJECT(obj1),
                 "anevent",
                 G_CALLBACK(demo_obj_cb_an_event),
                 NULL
);
```

Imagine a class ”Demo::Bar” with a signal ”anevent” that needs to be renamed. From the source code, tools can not deduce whether ”anevent” in the ”g_signal_connect” call needs to be renamed or not. However, before such a call, code can be inserted to ask for runtime type information of object ”obj”. If the type of ”obj” indeed features the signal the developer wants to rename, this function call can be marked for ”renaming the signal name”.

The same idea can be used for ”g_signal_emit_by_name” (certainly in case of action signals, so called named virtual function members), ”g_signal_connect” for notify, g_object_{set|get}, and so on ...

Inserting such functions that asks for runtime type information may be accomplished by using aspects, known from Aspect Oriented Programming (AOP) [KLM+97]. Currently, there is research for AOP in C: AspectC [CKFS01].

7.5 Summary

There are very few refactoring tools for C. Because of some problems inherent to C, such as the CPP, refactoring tool support is harder to write for C than for e.g. Java. However, the Xrefactory and especially the CRefactor projects provide refactoring frameworks for C. Provided that refactoring in C incorporating the CPP is possible, Glib-C refactorings are feasible to carry out as they are merely chainings of ”standard” C refactorings, driven by the Glib-C idioms.

In Glib-C the rename function member is e.g. easier to carry out, than in C++ thanks to the idioms. In C++ the static type of each object to be known, while in Glib-C function members of a certain static type are easily recognized through their prefix. In Smalltalk-80, dynamic refactoring is even required to successfully rename function members. In this last case, the refactoring is only as good as the regression test suite and thus may not be always 100% complete.

However, in case of GObject properties and signals, Glib-C needs some of the ideas of dynamic refactoring too. However, guided refactoring may be applied too. This would rename the signal or property itself, but not its usages. For this, the tool searches for usages of properties and signals with that name and interactively asks if the found usages need to be renamed or not (which is of course less fool-proof as using some of the ideas of dynamic refactoring).

Luckily, in many cases, some heuristics can be applied. In most cases the static type can be determined, excluding some usages to be renamed.

```
GtkEntry *entry = ... ;
g_signal_connect(G_OBJECT(entry), "activate", ...);
```

However, some of the idioms are required to at least deduce ”cornerstone” information. E.g. to recognize a Glib-C class. Other idioms are more mere coding conventions. However, when they are not correct some refactorings will not be completely successful. E.g. when renaming a data member, the setter/getter functions must have a logical name. E.g. in Eclipse and Java likewise idioms are used. If they are not correctly applied, Eclipse cannot refactor the setter/getter successfully along side the renaming of the data member itself.

As already been noted in chapter 5, Glib-C knows the common concepts of OO and these can be deduced from the source code based upon a set of idioms. Because of this, we can also reason about OO refactorings in Glib-C. Although C has no concepts of class hierarchies, we can still talk about ”push up function member” in Glib-C. As in many other OO environments, the same pre-conditions must be met to successfully apply the refactoring. The information for these pre-conditions is also deduced from the source code thanks to these idioms and we can thus reason about refactoring as a common applied OO concept.

Chapter 8

Overview

8.1 A Critical Look

In this section we take a critical look at Glib-C in general, and focus on the issue of runtime introspection and idioms.

8.1.1 General

The GType and GObject modules of the Glib library provide Object Oriented features for C, which C itself lacks. By using idiomatic programming, developers are able to express their OO designs in a coherent, usable and understandable system. These idioms are defined in this thesis in such a way that as much information could be deduced from the code. E.g. although C lacks any concept and syntax for a class (or any other OO feature for that matter), a tool must be able to recognize GType/GObject based classes in C. The applied idioms are called ”strong”. GType/GObject based C code using these strong idioms, is referred to as Glib-C.

However, these idioms emerged from the already existing naming conventions (actually a set weaker idioms) that existed in todays GType/GObject based C code, such as the GTK+ and GNOME projects. This means that the herein defined idioms are certainly not perfect.

E.g. the strong idioms put a lot of restrictions on the package name. A package name should only contain one capital at the start, prohibiting names as ”XmlLib”. Using other naming conventions, these restrictions would not have been needed. E.g. we could use

```
void Demo_LuxeCar_setTax(Demo_LuxeCar *self, guint tax);
```

instead of

```
void demo_luxe_car_set_tax(DemoLuxeCar *self, guint tax);
```

This kind of naming conventions take away the restriction on the package name ”Demo” as it now could be e.g. ”DemoLib”. I.e. in this case, the only restriction is that the underscore cannot be used in the package name (and class name and function member name for that matter). It would also be easier to recognize package name, class name, and function member name from this C function name and the object structure name . It also would allow nested packages such as ”Demo::Cars::LuxeCar” (where thus the underscores in the function member and object structure names merely represent scope operators). In this case we could have something like:

```
void Demo_Cars_LuxeCar_setTax(Demo_Cars_LuxeCar *self, guint tax);
```

In our quest to create tool-support to be able to build and comfortably grow software in Glib-C, we discovered that the strong idioms were sometimes too strong for what was needed. Weaker idioms sometimes sufficed to deduce the required information and to work with that information. In reality some of those idioms are probably to weak to be practical. Nevertheless, some of the lessons learned could be applied to the strong idioms (making them less strict in some areas). Striking examples of these, i.e. that were practically most useful, were related to the prefix of the function members, and the naming conventions regarding Glib-C interfaces.

E.g. we unleashed our tools on the GTK+ 2.2 library. In most of the cases the prefixes were conform to the idioms. However, if this was not the case, we could still recognize e.g. the function members of the Glib-C class/interface, by inspecting the so called Get-type function. The only requirement was that there was a ”GType <aprefix>_get_type()” function declared in the header file. The prefix of this function is used as the prefix of the function members. Using this information, we could in all cases find the function members of identified Glib-C classes.

Another example is, as been noted, related to Glib-C interfaces. The type of the interface structure should be named ”<TypeName>Iface” to be recognized, if we would rely on the strong Glib-C idioms. However, since a Glib-C interface structure has as first member a ”GInterface” (all Glib-C interfaces inherit from GInterface), such a structure could be identified (with the extra help of some heuristics) as a Glib-C interface, even if the name of the interface structure deviated from the idioms.

In this thesis we made a clear distinction between "used libraries" and "project code". In the case of "used libraries" we could rely on runtime introspection to deduce a lot of information, even with very weak idioms.

These last issues are a consequence of the technical side of Glib-C, which we shall not discuss further. However, the technical side of Glib-C is an important aspect in being able to deduce information, as e.g. interfaces indeed must inherit from the GInterface structures. This allows us to identify Glib-C interfaces rather quickly. Also, the Get-type function trick is actually a consequence of the technical side (because Glib-C needs such a function for every type to work properly). It is also this Get-type function that allows us to use runtime introspection. However we still require the idiom that the Get-type function ends on "_get_type".

8.1.2 Runtime Introspection

Runtime introspection plays an important role in being able to deduce the required information: even in the case of "used libraries", which respect the strong Glib-C idioms, runtime introspection is still needed. Tools must rely on it to retrieve some important information, such as what interfaces are implemented by a class, or which prerequisites an interface has. This itself is an important issue, but more importantly is that we have to deal with the technical side of runtime introspection itself, such as speed, feasibility and practical use.

In the "Glib-C type browser" (see figure 8.1), a prototype that has not been discussed up till now, we approached information deduction by first recognizing the Glib-C type using the strong idioms (incorporating some of the weaker ones, see next subsection). Next, the tool creates a small program in which only 3 variables are different from type to type. Those three variables are

- the header file declaring the type to be included,
- the specific Get-type function, and
- an initialization function for the used library

The first two elements are clear. The latter, the initialization function, is normally "g_type_init". This function initializes the Glib-C type system after which it can be used. This function is thus normally the first one that is called in a Glib-C based program. However, some projects require other initialization functions, such as "gtk_init" for the GTK+ UI toolkit and "gst_init" for the GStreamer graph-based multimedia framework. Besides these 3 variables, we also need to know how to compile the small introspecting program, as different types may need to be linked with different libraries. Next, the small program is compiled and executed, after which it will return information about the inspected type.

In reality, such as in a full blown IDE, the compile flags of the project in the IDE could be used for the compile flags of the introspecting program. (However, this is probably overhead, but this is certain to work). The header file to be included in the introspecting program and the Get-type function of the type to be introspected, is of course known. Only the initialization function must be custom specified if that is needed. This can be done per package (namespace), such as "Gtk" for GTK+ and "Gst" for Gstreamer. If the initialization function is not specified, the standard type system initialization function is used.

However, it would have been desirable that the tool could deduce the required information about Glib-C types without runtime introspection. Actually, this is a consequence of the technical side of Glib-C. The GType/GObject type and object system chooses a way to implement interfaces, properties and signals. Nevertheless, this is something that might be investigated: yes, the system works, but could an equal or even better system be provided in such a way, that tools (or the reader of the Glib-C code for that matter) can deduce information such that no runtime introspection is required?

Nevertheless, runtime introspection is enabled by building a small program and executing it. This seems not very ideal because of the issue of speed. However, we were surprised how relatively fast it actually was. Especially assuming that a modern IDE requires a solid and fast computer, runtime introspection actually is acceptable. Certainly when the type is inspected only once and only when needed. After wards this information can be cached and made persistent between sessions. (And since we are dealing with used libraries which are static in the project, this itself is not a problem).

Also, using runtime introspection is the right way to go if dealing with used libraries in combination with weaker Glib-C idioms (in case of e.g. older existing libraries). In this case the tools are using runtime introspection for finding and creating class hierarchies anyway, and are our prime source of information (instead of providing use with additional information). Also Java environments use something similar in case of libraries, i.e. reflection.

8.1.3 Weak and Strong Idioms

There is a clear distinction between "used libraries" and "project code". The former are libraries that are used in the project, and tools only have the header files at their disposal to extract information from. The code of the system that is in development, is referred to as "project code". In this case, the complete sources are available.

In case of "used libraries", we basically have to know which types these libraries offer. I.e. a type browser must be able to create a class and interface hierarchy and view information about the classes and interfaces. However, besides identifying what types are available in the library, tools also need to identify some of the elements that make up a Glib-C type. For code generation we basically need three things of each Glib-C class: the type macro (or the Get-type function itself), the object structure and the class structure. This information is needed to inherit from the base class. I.e. the first elements of the object and class structures are the object and class structures of the parent class respectively. Also, the parent is registered in the Get-type function of the class. For this we need the Get-type function or type macro of the parent class.

In the case of "project code", tools of course also need to know what types are declared, but this time they need a lot more information about the elements part of the Glib-C type. E.g. for refactoring, we basically need to find and identify everything.

We now know what kind of information needs to be deduced from either "used libraries" or "project code". Next, lets take a look at which idioms are required to do so.

In the case of "used libraries" very weak idioms could suffice. The first thing that is required is that of course the Glib-C type is recognized. This is obviously the cornerstone for information deduction. This can be achieved by looking up functions of the form "GType <aprefix>_get_type()". Those functions possibly represent a Get-type function of a Glib-C type. If this function indeed allows runtime introspection, tools know they found a Glib-C type. Using runtime introspection itself, tools deduce whether it is an interface or a class. In the latter case, it will e.g. retrieve information such as the type name, the parent, implemented interfaces, and so on.

There is only idiom required: there must be Get-type function of the given form. However, tools also need to know what the object and class structures are from this class. For this, we again need some idioms. Tools can search for a structure with the type name of the found Glib-C type. After this, it can search for another structure with the type name appended with "Class". Remember that the strong idioms also put some restrictions on the type name, but in this case, this is not needed (to identify types).

However, if tools want to be able to extract package and class name (or interface name) from the type name, the strong idioms must be correctly applied to this type name. This is actually needed if tools must find the type macro of the corresponding Glib-C type. However, in chapter 7 we have discussed alternative (but less safe) ways to find the macro. Nevertheless, if the macro cannot be identified the Get-type function can be used instead.

For applications as type browsing, hinting for properties and signals, method completion, ... information acquired by runtime introspection suffices. However, we still need to know the function members. These can be found by looking at the prefix of the Get-type function. Again, this is a reasonable weak idiom.

Next we take a look at the idioms required in the case of "project code". There we can not rely on runtime introspection, as code must not be always compilable to e.g. enable method completion. In this case, tools again can rely on very weak idioms, in which the Get-type function again plays an important role. In an extreme case, the Get-type function can be used to deduce what the object and class structures are, even if they do not convey to the "TypeName" and "TypeNameClass" rules respectively. However, on the other hand some strong naming conventions are required to find e.g. the private data structure.

However, since project code becomes a used library at some point, we cannot allow such very weak idioms, as in the extreme case we just discussed. Also, if weak idioms are applied, the convenience macros may be found but how are renamings correctly applied?

Finally and of utmost importance, the point of idioms is that the code is readable and understandable. Also, with the correct idioms applied, the developer knows readily what e.g. the type safe casting macro is. As been noted before, Glib-C can only be spoken correctly and fluently, if a certain set of rules are respected.

I.e. weak idioms may be sufficient to deduce large and important parts of the required information, strong idioms make things easier and are just plain better. However, the weak idioms are useful for e.g. dealing with existing libraries, or when an existing (not Glib-C conform) GType/GObject based C project is imported in the tool-suite. However lessons learned from the weak idioms may be applied to the strong idioms, making them less strict and relaxed on some areas (such as we have discussed in section 8.1, e.g. allowing a deviated prefix for function members). This may suggest we can define different levels of Glib-C idioms.

However, there is a last issue that is not yet discussed until now. Even with very strong idioms, tools must sometimes "read" code anyway. Code reading is slower than looking up names in a cross reference: e.g. Glib-C classes can be identified by searching names of structures, but signals can only be identified by "reading" the class initialization function.

8.2 Enforcing Idioms

8.2.1 General

Tools rely on the idioms part of Glib-C. However, this can only be safe if the tools can trust these idioms. Imagine e.g. a refactor browser part of an IDE. The Glib-C code is being developed in that closed environment. Refactoring has to rely on the idioms of Glib-C: it can only successful refactor if some idioms are met (e.g. imagine the corresponding convenience macros, the private data structure, ...). However, for this the idioms must be enforced by the same closed environment. I.e. idioms can only be trusted if they are enforced. However, the question is, is it possible to enforce the idioms and if so, how?

There are different situations to be considered. E.g. when using a class/interface generator, the tools flag the resulted code as representing a Glib-C type. From there on, the tools can maintain strict control over the used idioms. Another case is when the developer starts from scratch by hand, and the tool must recognize a Glib-C type from this manually written code. Importing existing GType/GObject based code in the closed development environment is a last possible situation. However we assume the tool-suite knows that a certain source file and header file declare and define a Glib-C type.

8.2.2 Idioms Enforcement

First the tool needs to enforce that **one and only one Glib-C type is declared and defined** per header and source file duo. Lets focus on classes. In this case the tool-suite looks at "ATypeName" and "ATypeNameClass" structures in the project code (whose first data members are

"AnotherTypeName" and "AnotherTypeNameClass" respectively). If there is exactly one such duo found, then the tool-suite is satisfied. Otherwise, the tool-suite will not allow the code to be regarded as valid. However, it is possible that there is another Glib-C class declared and defined, but not meeting the Glib-C idioms. Using a weaker set of idioms or a method like "Rollback to GObject(Class)" the "intruded Glib-C class" can be found. In practice, this check may run in the background to avoid slow responsiveness of the tool-suite.

The object and class structures are regarded as the cornerstone of the Glib-C class. The object structure should of course be at least declared in the header file. For now, we pose no further restrictions.

Next, the tool-suite requires the presence of a corresponding **Get-type function** of the form "GType <aprefix>_get_type()". In case of weaker idioms, in which we allow a deviated prefix (from the strong idioms), tools can "read" the Get-type function and confirm if the object and class structure are registered with the GTypeInfo structure. However, if strict strong idioms are used, the job is simplified: for one, TypeName must correspondent to the exact rules. If not, the tool-suite can flag a warning and refuses to go on. From this correct TypeName, the prefix can be deduced and then there must be a "GType <strictprefix>_get_type()" function present.

In the Get-type function there is a registration call. First it must be checked if the registered TypeName is the name of the object structure. Next, the parent type macro can be checked. If these are valid, tools can deduce whether the class is registered as abstract or not. If not, there must be a **new operator function** declared of the form <prefix>_new[_with_...]([...]);

The next step is to enforce the presence and the naming conventions of **the convenience macros**. As with the prefix, we take a look at the case of very strict idioms applied and in the case of weaker idioms. If custom "<PACKAGE>" and "<CLASS>" names are allowed in the convenience macros, the tool can search for "<A>_TYPE_<B>" and "<A>_IS_<B>". From there on it can oblige the presence of the other macros with consistent naming. In case of very strong idioms in which the "TypeName" allows us to deduce the "<PACKAGE>" and "<CLASS>" values, the job for the tool-suite is of course simplified because these values can be directly deduced from "TypeName".

The object and class initialization functions are optional. However, in the GTypeInfo structure that is registered in the Get-type function, the tool can easily look up if there are such initialization functions registered. If this is indeed the case, the tool can check for the correct naming conventions on these functions.

Next, we take a look at **virtual function members**. Virtual function members have function pointers in the class structure defined. If these function pointers are not registered as a function pointer for a default object signal handler, then the tool-suite can require to have a corresponding function member wrapper conforming to "<theprefix>_<funcpointername>". This is essential for recognizing virtual function members and when virtual function members are overridden/extended.

If virtual function members are available, the tool-suite must oblige the developer to define the class structure in the header file. If this is not the case, then the virtual function members are useless as they cannot be overridden.

Signals are recognized by their registration in the class initialization function. However, remember that conditional compilation directives make things difficult. In this case the tool-suite may require default values for macro values used in the conditional compilation. The same problems are relevant for **GObject properties**.

Since signal registrations have to be read anyway, enums for these signals are easily associated. Also, property registrations have to be read for information deduction (e.g. type browsing), thus enums can be associated easily. The tool-suite can from there on enforce idioms.

For every found GObject property there must be a corresponding **data member**. E.g. for property "luxe-tax" or "luxeTax" there must be a data member "luxeTax". However, there is problem with identifying data members as there is a problem with the **private data structure**. This structure can only be identified through strict naming conventions. Heuristics can search for such a structure, but will not always succeed in finding one. This means that idioms cannot always be enforced. In Glib-2.4 however, it is possible to register the private data structure with the type (in the Get-type function). Using this possibility, the private data structure can easily be identified.

Not only the private data structure cannot be easily enforced to convey certain idioms, but also something more essential: the **function members**. Since function members are only associated through their prefix with the Glib-C type, it is thus the prefix we solely rely on to identify function members. However, if the correct prefix is not applied, the tool-suite will not identify any function members, hindering things like method completion, automatic unit test generation, refactoring, type browsing and so on. I.e. the developer is rewarded with the comfort of the tool-suite if he or she use the correct naming conventions. This is of course an essential idea as it stretches to all areas of the idioms.

However, virtual function members can be found (see earlier). Also, if function members are identified and further recognized as **getters/setters**, it must find corresponding data members or it will complain.

8.2.3 Overview

A great deal of idioms can be enforced. However there are some problems. Most notably the private data structure and function members cannot be enforced. However, if the developer does not use the corresponding naming convention, the tool-suite will not offer any support for them. I.e. the tool-suite can only deliver comfort and productivity if the developer follows some naming conventions. However, using a consequent prefix for the function member is a cornerstone for ADT style programming, and is a very basic rule. In reality, we have not encountered any class that broke that rule.

Luckily, the cornerstone of the idioms that at least allow tools to identify types, can be enforced. And while some important things cannot, more advanced idioms are also able to be enforced on the code. E.g. in case of virtual function members, all related idioms can be enforced, allowing tools to locate them, identify when they are overridden and so on. I.e. allow everything what is required by our prototypes, enabling them to function correctly.

8.3 Glib-C and OO concepts

8.3.1 OO in Glib-C

In our quest to create tool-support for Glib-C, we used common OO concepts, although C itself lacks any knowledge of these. However, Glib-C enables us to speak about them. Whether we were building tool-support for code generation, testing or refactoring, it was with terms and phrases like ”package name”, ”class name”, ”parent class”, ”function members”, ”data members”, ”interfaces”, ”inheritance hierarchies”, ”overriding function members”, and so on we were dealing with.

Indeed, it was because Glib-C could represent these concepts we were able to build tool-support. How these concepts look like at the level of code does not matter that much. Whether the concepts are represented with syntax as in C++ or with idioms as in Glib-C, is not important, as long as we are able to deal with them. Indeed, it are the common OOPLs that actually model the same concepts over again. They may have differences in their extra features and especially their purposes, but all the common OOPLs are viable alternatives on the level of OO to each other, as we consider the OO concepts they are able to model.

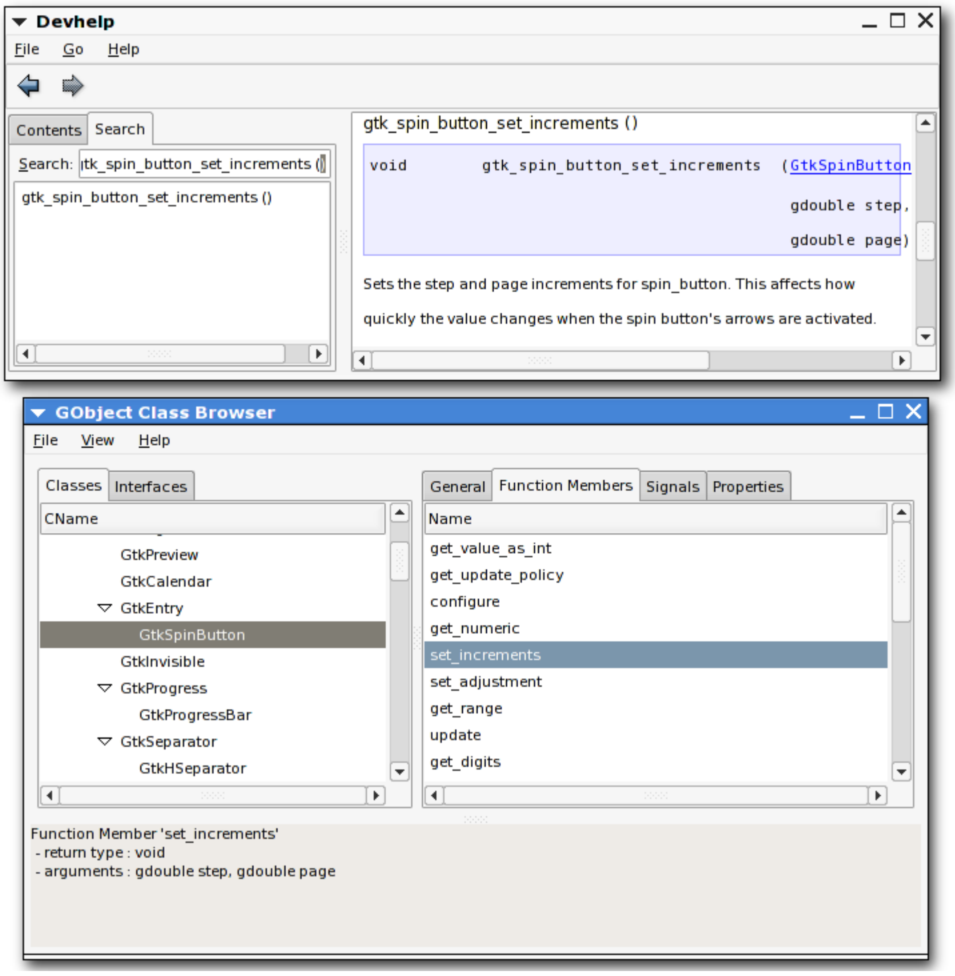
The fact that we can model these concepts language independent of how the language looks like became e.g. very clear with code generation. In a comparison between Glib-C and C++ we noticed that a Glib-C class stub was 5 times as big as a C++ class. Judging on merely on the level of the language, Glib-C seems to be a bad OO alternative to a language such as C++. However, in both languages we deal with the same concepts. When working with these concepts, we have as much work to do in both languages as we have to deal with the same amount of the same concepts. Whether these concepts are called package or namespace, function member or method (as in Java), they represent the same idea. In the end, we had has much work to do in both environments, although the final resulting code was very different.

Indeed, we have to map concept to the code. This was done by the code generator (figure 5.1) we just discussed. Of course, a viable OO platform allows to retrieve the concepts from the code. It is very clear that tool-support relies on this. Luckily this is possible as e.g. our type browser illustrates (figure 8.1). For this we relied on the idioms. In the case of libraries we also had to rely on runtime introspection. Again, how the concepts are represented is not important, as long was we can retrieve them. In Java e.g. reflection is used in the case of libraries.

8.3.2 Meta-systems

It is clear that the concepts of OO are of utmost importance. We constantly work on a higher level. Again, whether certain elements are called function members or methods, they model the same thing. Therefore we could independent of the language, the underlying terminology and system, create a meta-model. Our tools used a e.g. a simple custom created data layer to represent class hierarchies and their specific information.

This is of course what UML exactly does. UML is the Unified Modeling Language specified by OMG. The UML architecture provides a (object) meta-model to represent common concepts like classes, interfaces, operations (function members), generalization (inheritance) and so on. UML is perfect



gtk_spin_button_set_increments ()

gtk_spin_button_set_increments ()

```
void        gtk_spin_button_set_increments   (GtkSpinButton
                                              gdouble step,
                                              gdouble page)
```

Sets the step and page increments for spin_button. This affects how quickly the value changes when the spin button's arrows are activated.

GtkPreview
GtkCalendar
GtkEntry
GtkSpinButton
GtkInvisible
GtkProgress
GtkProgressBar
GtkSeparator
GtkHSeparator

get_value_as_int
get_update_policy
configure
get_numeric
set_increments
set_adjustment
get_range
update
get_digits

Function Member 'set_increments'
- return type : void
- arguments : gdouble step, gdouble page

Figure 8.1: A last undiscussed prototyped tool: a Glib-C type browser. This tool symbolizes the ability to model the concepts from the code. The DevHelp tool is a gtk-doc documentation browser. The type browser uses this tool to show documentation about the selected item.

applicable to represent e.g. design of a system using class diagrams. However, not only the concepts of OO on the level of design are common, but also e.g. on the ideas that surround OO, like refactoring.

As been discussed, refactoring offers typical operations for adding, changing, removing and moving items in the hierarchy. These are e.g. "pull up method", "extract method" and so on. Also, these refactorings are formulated independent of the underlying language. E.g. for a "pull up method" we just require to have classes which could inherit from other classes, and classes must have methods (function members). Besides that, in refactoring some preconditions must be met before the refactoring may be applied. Again, we encounter the same concepts and questions, independent of the language. E.g. one of the preconditions for renaming a method of a certain class is of course if there is not already such a function member with the new name in the class or its parents? Again, we ask ourselves questions about specific design.

However, UML does not suffice as meta-model for refactoring, as e.g. a "pull up method" requires that the method is not accessing data members of that specific class. To check this we need a code representation and UML is not sufficient for this [GSMD03]. However, there are meta-models such as FAMIX or an extension to UML called GrammyUML [GSMD03]. These meta-systems are developed with the intend not only to represent design but also to allow refactoring on that meta-level.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

The question carried through this thesis, is whether Glib-C is fit as a viable OO alternative. It is clear that Glib-C knows the concepts of OO. However, Glib-C as a tool on its own, is not a very productive and ideal environment from an OO point of view. The criteria for an viable OO platform are that the developer has a certain level of comfort and is able to grow software. To enable this, we tried to create tool-support for Glib-C. This proved to be possible, because we could exploit the concepts of OO, that are available in Glib-C through the usage of idioms.

The resulted tool-suite in where the language is only a mere tool, allows the developer to grow software. Which elements of the tool-suite offered abstractions for certain concepts of the OO paradigm, does not matter much. Important is that independent of how responsibilities in the tool-suite are distributed, we can work and play with those concepts.

This means that (tool-supported) Glib-C is a viable OO alternative. This does however not mean that it is a perfect substitute for e.g. Java projects. On the level of OO they provide likewise concepts and both allow to work with them on a productive way. However, there are more criteria to be considered than the OO power of a platform. Glib-C may be perfectly applicable for system libraries while Java is more fit for end user applications.

9.2 Future Work

This thesis focused on the questions whether Glib-C is a viable object oriented alternative. However, there are some future work related to Glib-C.

- UML, GrammyUML, FAMIX, ... are targeted for use in combination with common OOPL languages. However, how do they interact with environments that use idiomatic OO programming? Are there some specific problems? How do e.g. Glib-C and GrammyUML relate to each other?
- Investigate the practical feasibility of our created prototypes. An example of this would be to create a scanner based solely on weak idioms and runtime introspection. This scanner would be tested in a small IDE application with type browsing, real method completion, and so on. It would also be interesting to test whether code reading is feasible: e.g. even if the implementation functions are not conform to the strong idioms, they can be found by looking at the assignments of functions to the function pointers part of the virtual table. However, this may be slow and unrealistic in a real-life tool-suite.
- The refactor browser we created uses hacks like the usage of "Xref" utility, and assumes no problems with the CPP. However, researching refactoring in Glib-C based upon using e.g. the CRefactory project, integrated in a real-life usable refactor browser could prove to be interesting.

Chapter 10

Appendix A: Glib-C Idioms

The Glib-C idioms are described in this appendix. Therefore the next terminology is used:

- must : obliged
- should : highly recommended, but not obliged
- must not : not allowed, still doing so will break the idiom
- may : it is allowed

To denote type names, we use the notation from C++. E.g. ”Demo::Car”. This is a class ”Car” in package (a.k.a. namespace) ”Demo”. ”Demo::Car::getName()” denotes function member ”getName” from that class.

10.1 Introduction

This section explains the used idioms in Glib-C. This will be done by example. C code (Glib-C class/interface declarations/definitions) not conforming to those idioms is not considered Glib-C code.

In Glib-C we have the concepts of packages, classes, interfaces, data members, properties, function members, virtual function members and signals. Table 10.1 shows example values for these concepts. This example will be used throughout this appendix. The corresponding Glib-C class declaration and definition is shown in appendix B.

Concept	Value
Package	Demo
Class	Car
Parent Typename	GObject
Data members	name mileage
Properties	name mileage
Function Members	get_name set_name get_mileage set_mileage
Virtual Function Members	drive
Signals	driven

Table 10.1: Values for a Glib-C class example "Demo::Car"

10.2 Identifiers

In Glib-C there is a **package name** and a **class name**. A class must always be declared in a package, i.e. a package name is obliged. A package name must start with a capital and should not contain any other capitals. A class name must start with a capital and may contain other capitals. Succeeding capitals in a class name should be avoided. The **type name** is a concatenation of the package name and the class name.

Example, the Glib-C class "Demo::Car":

- package name : "Demo"
- class name : "Car"
- type name : "DemoCar"

Package names like "DemoPackage", which contains a second capital, is highly discouraged.

In case of Glib-C interfaces there is a package name and an **interface name**. The same rules of class name apply to the interface name. The type name is in this case the concatenation of the package name and the interface name.

10.3 Structures

A Glib-C class declaration compromises 3 C structures: The **instance structure** or **object structure**, the **class structure** and the **private data**

structure. The instance and the class structure are obliged. They must be declared. The private data structure is optional.

The name of the type corresponding with the instance structure is the type name of the Glib-C class. The name of type of the class structure is the name of the instance structure (i.e. the type name) appended with ”Class”. The name of the type of the private data structure is the name of the instance structure appended with ”Private”.

Example ”Demo::Car”:

- type name : ”DemoCar”
- instance structure name : ”DemoCar”
- class structure name : ”DemoCarClass”
- private data structure name : ”DemoCarPrivate”

The C structure itself is has the name of the corresponding type, prefixed with an underscore. The typedefs should be done separately as in the example (see appendix B). Although these typedefs may be declared as in the next example (but this is certainly not recommended):

```
typedef struct \_DemoCar {
     /**/
} DemoCar;
```

A Glib-C class always inherits from another class. The instance and class structures contain as first member the instance and the class structure of the corresponding parent class. those members should be named ”parent”. See lines 25 and 33 in the header of the example.

If a private data structure is present, the instance structure should contain a pointer to the private data structure. This pointer should be named ”priv” or ”_priv”, the last one is recommended. This pointer should be the last data member in the instance structure (line 29 in the example).

Data members in the instance structure, most likely public data members, should start with a small letter and should not be prefixed with an underscore. The first letter of each internal word must capitalized. Private data members in the instance struct (in the case there is no private data structure) may be prefixed with an underscore. Example of an instance struct (Demo::Object):

```
struct \_DemoObject {
    GObject parent;

    /* public datamembers */
    gchar *publicDatameberOne;
    guint anotherPublicDatamember;

    /* private */
    DemoObjectPrivate *_priv;
};
```

Shared data members are placed in the class structure. The same naming conventions as for data members in the instance structure apply. A private shared data structure may be created but is not common. Such data structure must be named e.g. "DemoCarClassPrivate" or "DemoCarClassPriv".

Function pointers and pointers to default signal handlers, which are part of the class structure, are discussed in their corresponding sections. However, here is an example of class structure (Demo::Object):

```
struct \_DemoObjectClass {
    GObjectClass parent;

    /* shared datamembers */
    gchar *aPublicSharedDatamember;
    guint *_aPrivateShardDatamemmber;

    /* no signals, no vtable */
};
```

In case of a Glib-C interface declaration, 2 structures are involved: The **interface structure** and a **dummy object structure**. The interface structure is the equivalent of the class structure and the dummy object structure is the equivalent of the instance structure.

The type of the dummy object structure is actually only a typedef for a non existing structure. The name of the type of the dummy object structure must be the type name of the interface. This is exactly the same in case the of an instance structure of a Glib-C class. An example for an interface (Std::XmlSerializable):

```
typedef struct _StdXmlSerializable StdXmlSerializable.
```

The name of the type of the interface structure must be the type name of the interface, appended with "IFace". E.g.

- package name : Std
- interface name : XmlSerializable
- type name : StdXmlSerializable
- dummy object structure name : StdXmlSerializable
- interface structure name : StdXmlSerializableIFace

As in the case with Glib-C class related structures, the names of those structures must be prefixed with an underscore and it is advised to "typedef" them separately. E.g.

```
typedef struct _StdXmlSerializableIFace StdXmlSerializableIFace.

struct _StdXmlSerializableIFace {
    GTypeInterface iface;

/* ... methods here ...*/
};
```

10.4 Convenience Macros

A Glib-C class or interface declaration should be accompanied with convenience macros for type retrieval, type safe casting, type checking and class/interface access from an instance. Those macros must exhibit some exact naming conventions. They must be in uppercase and all start with the name of the package. Each internal word in the macro must start with an underscore.

Type retrieval macro The type (retreival) macro conveys to <PACKAGE>_TYPE_<CLASS>. <CLASS> is actually the class name (or interface name in cases of interfaces) in uppercase, where the name is splitted on every new non succeeding capital, concatenated with an underscore. E.g.. for a class name "Car",the value of <CLASS> equals to "CAR". With class name "FemalePerson", <CLASS> equals to "FEMALE_PERSON". A special case is a class name like "XMLReader". This class name leads to a value "XML_READER" for <CLASS>. I.e. the class name is splitted before the last capital in the succeeding array of capitals. Though, class names with succeeding capitals should be avoided. A better name for class name "XMLReader" should be "XmlReader".

Type save casting macro This macro conforms to <PACKAGE>_<CLASS>.

Type checking macros To check an instance struct to be of a certain type the macro <PACKAGE>_IS_<CLASS> is used. To check a class structure to be of a certain type the macro <PACKACKE>_IS_<CLASS>_CLASS is used.

Class retrieval macro This macro conforms to <PACKAGE>_<CLASS>_GET_CLASS.

Interface retrieval macro This macro conforms to <PACKAGE>_<CLASS>_GET_IFACE.

10.5 Function members

Function members must be all in lowercase. Every internal word in the function member is separated by an underscore. Function members start with the package name, followed by an underscore and the class name. In this case the class name conveys to the same rules as in the convenience macros. I.e., they must have the same format as <CLASS>, except it is in lowercase. This is referred to as <class>. The same goes for <PACKAGE>", which we refer to as <package>. <type> is used to denote "<package>_<class>. Examples:

```
/* for a class Demo::Car */
demo_car_set_name
demo_car_drive
/* for an interface Std::XmlSerializable */
std_xml_serializable_write_to
```

The first parameter of a function member must be a reference to the object structure. The type of this parameter should be a pointer to type of the instance structure. The argument should be called "self". E.g.

```
/* class Demo::Car::drive */
demo_car_drive(DemoCar *self, guint distance, gboolean backward);
```

In case of interfaces the first parameter must be a pointer to a dummy object structure. The first parameter should be called "self". E.g.

```
/* interface Std::XmlSerializable */
std_xml_serializable_write_to(StdXmlSerializable *self, StdXmlStream *stream);
```

Virtual function members are function pointers in the class structure. Normal function members wrap the call of such a function pointer. The name "virtual function members" are interchangeably used for the function pointers and the function member wrappers. They are closely linked together.

Those function member wrappers convey to the same naming conventions as for any other function member. Actually, "demo_car_drive" that has been discussed before is a virtual function member (see appendix B). The function pointers have the same name but without <type> prefixed. E.g. the name of the function pointer of the virtual function member "Demo::Car::drive" is "drive".

The implementation functions that are being pointed by the function pointers should be called <type>_impl_<function_member_name>. <type> should correspondent with the Glib-C class were the implemenation is defined. Since these functions must reside in the source file, it is advised they are declared "static". E.g.

```
static void
demo_car_impl_drive(DemoCar *self, guint milage, gboolean backwards) {
}
```

Shared function members or class-wide function members, also known as static function members in C++, are exactly as normal function members. However, after <type> "_class" must be appended. Also, instead of the object structure being passed as first argument, the class structure must be passed. This first parameter should be called "klass". The word "class" may not be used since this is a C++ keyword. E.g.

```
guint demo_car_class_get_instances(DemoCarClass *klass);
```

10.6 Registration, initialization and destruction

Registration of a Glib-C type (class or interface) happens when it is instantiated for the first time. This is done using the so called **Get-type function**. This function must be called <type>_get_type() and no other way. E.g.

```
GType demo_car_get_type();
```

Initialization of the object structure is done either using the "constructor" virtual function member introduced in the common base class "GObject" or using the **instance initialization function** associated with every Glib-C class. The latter is most common and is advised. This initialization function is registered in the Get-type function of the Glib-C class. Initialization of the class structure is done using the **class initialization function** which is also registered in the Get-type function.

When using the constructor virtual function member the parent implementation must be called first. When using the initialization function associated with the GType, the type system automatically chains up. The latter is thus more safe.

The naming conventions for the initialization functions convey to "normal" function members, i.e. "<type>_init" and "<type>_class_init" for the object and class structure respectively.

Destruction of an instance of a Glib-C class is implemented by overriding the "dispose" and "finalize" virtual function members introduced in the "GObject" base class. The naming conventions of those correspondent with any other virtual function member implementation: they must be called "<type>_impl_dispose" and "<type>_impl_finalize" respectively.

If the class is not abstract, a **new operator function** must be provided. This function has the form "<type>_new". E.g.

```
DemoCar* demo_car_new();
```

In the case that the new operator takes arguments, the form "<type>_new_with_<args>" is used. E.g.

```
DemoCar* demo_car_new_with_name(gchar *name);
```

If the class is abstract, no new operator function may be specified.

10.7 GObject Properties

GObject properties consist of 3 things: the corresponding data members with optionally getters and setters, an enumeration, and the "GObject::set_property" and "GObject::get_property" virtual function members. When specifying GObject properties they must be registered in the class initialization function. The GObject::set/get_property virtual function members should also be overridden.

Those virtual function members convey to the standard virtual function members and should be named as in the example.

The corresponding enumeration names must be PROP_<PROPNAME> or <PREFIX>_PROP_<PROPNAME>. <PROPNAME> should be the capitalized version of the corresponding data member name. After choosing one of the two conventions, consistency must be applied. This way the property and the corresponding data member are associated.

GObject properties are also named by a string. The name must be either exactly the name of the corresponding data member or be a dashed representation of the corresponding data member. I.e. for a data member "canFocus" the corresponding property name must be either "canFocus" or "can-focus". The latter is more commonly used.

The next example shows how to install a property "name". The name of property "name" is exactly the same as the corresponding data member

```
g_object_class_install_property(
     gobject_class,
     PROP_NAME,
     g_param_spec_string(
         "name",
         "name",
         "The name of the car",
         "",
         G_PARAM_READABLE | G_PARAM_WRITABLE
     )
);
```

10.8 Signals

A signal has a name. The name of a signal does not compromise the name of the Glib-C class. E.g. the "Demo::Car" class has one signal called "driven".

GObject signals have a corresponding enum and array declared. The enumeration names must be the capitalized versions of the signal names. E.g for signal "driven" the corresponding enumeration name must be "DRIVEN". The array must be declared as <class>_signals. E.g.

```
static guint demo_car_signals[LAST_SIGNAL] = { 0 };
```

The name of the corresponding pointer to a possible default object handler, which is part of the class structure, must be exactly the name of the signal. Installing a signal on a type occurs in the class internalization function. This is done using "g_signal_new" and family. The name of the signal is passed as first argument.

E.g. the class Demo::Car has one signal named "driven" In the class structure there is thus a function pointer "driven", and the corresponding enum name of the signal is "DRIVEN".

```
/* global enum */
enum {
    DRIVEN,
    LAST_SIGNAL
};

/* in demo_car_class_init */
demo_car_signals[DRIVEN]= g_signal_new(
    "driven",
    DEMO_TYPE_CAR,
    G_SIGNAL_RUN_FIRST | G_SIGNAL_DETAILED,
    G_STRUCT_OFFSET(DemoCarClass, driven),
    ...
```

Chapter 11

Appendix B: Code Examples

11.1 Example: class ”Demo::Car”

11.1.1 Header file

```
#ifndef INC_DEMOCAR_H
#define INC_DEMOCAR_H

#include<glib-object.h>

G_BEGIN_DECLS

#define DEMO_TYPE_CAR                   \
         demo_car_get_type()
#define DEMO_CAR(obj)                           \
        (G_TYPE_CHECK_INSTANCE_CAST((obj), DEMO_TYPE_CAR, DemoCar))
#define DEMO_CAR_CLASS(klass)           \
        (G_TYPE_CHECK_CLASS_CAST((klass), DEMO_TYPE_CAR, DemoCarClass))
#define DEMO_IS_CAR(obj)                        \
        (G_TYPE_CHECK_INSTANCE_TYPE((obj), DEMO_TYPE_CAR))
#define DEMO_CAR_GET_CLASS(o)   \
        (G_TYPE_INSTANCE_GET_CLASS ((o), DEMO_TYPE_CAR, DemoCarClass))

typedef struct _DemoCar DemoCar;
typedef struct _DemoCarClass DemoCarClass;
typedef struct _DemoCarPrivate DemoCarPrivate;

struct _DemoCar {
        GObject parent;

        /*< public >*/
        /*< private >*/
        DemoCarPrivate *_priv;
};

```

```
struct _DemoCarClass {
        GObjectClass parent;

        /*< public >*/
        /*< private >*/
        /*< signals >*/
        void (*driven) (DemoCar*,gulong);

        /*< vtable >*/
        void (*drive) (DemoCar *self);
        void (*write_to_xml) (DemoCar* self, GString *output);
};

GType           demo_car_get_type();
DemoCar*        demo_car_new();

gulong          demo_car_get_milage(DemoCar*);
void            demo_car_set_milage(DemoCar*, gulong milage);
gchar*          demo_car_get_name(DemoCar*);
void            demo_car_set_name(DemoCar*, gchar* name);

inline void     demo_car_drive(DemoCar *self);
inline void     demo_car_write_to_xml(DemoCar* self, GString *output);


G_END_DECLS

#endif
```

11.1.2 Source file

```

#include "democar.h"

static GObjectClass *parent_class = NULL;

enum {
        PROP_0,
        PROP_MILAGE,
        PROP_NAME,
};

enum {
        DRIVEN,
        LAST_SIGNAL
};

static guint demo_car_signals[LAST_SIGNAL] = { 0 };

struct _DemoCarPrivate {
        gulong milage;
        gchar* name;
};


static void
demo_car_impl_write_to_xml(DemoCar* self, GString *output) {

}
```

```

static void
demo_car_impl_drive(DemoCar* self) {

}


void
demo_car_write_to_xml(DemoCar* self, GString *output) {
        if (DEMO_CAR_GET_CLASS(self)->write_to_xml != NULL) {
                DEMO_CAR_GET_CLASS(self)->write_to_xml(self, output);
        } else {
                g_error("Virtual␣function␣write_to_xml␣␣"
                        "has␣no␣implementation␣assigned");
        }
}


void
demo_car_drive(DemoCar* self) {
        if (DEMO_CAR_GET_CLASS(self)->drive != NULL) {
                DEMO_CAR_GET_CLASS(self)->drive(self);
        } else {
                g_error("Virtual␣function␣drive␣"
                        "has␣no␣implementation␣assigned");
        }
}


gulong
demo_car_get_milage(DemoCar* self) {
        return self->_priv->milage;
}

void
demo_car_set_milage(DemoCar* self, gulong milage) {
        self->_priv->milage = milage;
        g_object_notify(G_OBJECT(self), "milage");
}

gchar*
demo_car_get_name(DemoCar* self) {
        return self->_priv->name;
}

void
demo_car_set_name(DemoCar* self, gchar* name) {
        self->_priv->name = name;
        g_object_notify(G_OBJECT(self), "name");
}



static void
demo_car_impl_set_property(DemoCar* self, guint prop_id,
                        const GValue *value, GParamSpec *pspec) {
        switch(prop_id) {
                case PROP_MILAGE:
                        demo_car_set_milage(self, g_value_get_ulong(value));
                        break;

                case PROP_NAME:
```

```
				demo_car_set_name(self, g_value_get_string(value));
				break;

			default:
				G_OBJECT_WARN_INVALID_PROPERTY_ID (self, prop_id, pspec);
				break;
	}
}


static void
demo_car_impl_get_property(DemoCar* self, guint prop_id,
				GValue *value, GParamSpec *pspec) {
	switch(prop_id) {
			case PROP_MILAGE:
				g_value_set_ulong(value, demo_car_get_milage(self));
				break;

			case PROP_NAME:
				g_value_set_string(value, demo_car_get_name(self));
				break;

			default:
				G_OBJECT_WARN_INVALID_PROPERTY_ID (self, prop_id, pspec);
				break;
	}
}



static void
demo_car_impl_dispose(DemoCar *self) {
	G_OBJECT_CLASS(parent_class)->dispose((GObject*) self);
}


static void
demo_car_impl_finalize(DemoCar *self) {
	if (self->_priv) {
		if (self->_priv->name) g_free(self->_priv->name);
		g_free(self->_priv);
	}

	G_OBJECT_CLASS(parent_class)->finalize((GObject*) self);
}


static void
demo_car_init(DemoCar *self) {
	self->_priv = g_new0(DemoCarPrivate, 1);
}

static void
demo_car_class_init(DemoCarClass *klass) {
	GObjectClass *gobject_class = G_OBJECT_CLASS(klass);

	parent_class = g_type_class_peek(G_TYPE_OBJECT);

	klass->drive = demo_car_impl_drive;
	klass->write_to_xml = demo_car_impl_write_to_xml;

	gobject_class->dispose = demo_car_impl_dispose;
```

```
        gobject_class->finalize = demo_car_impl_finalize;
        gobject_class->get_property = demo_car_impl_get_property;
        gobject_class->set_property = demo_car_impl_set_property;

        demo_car_signals[DRIVEN]= g_signal_new(
                "driven",
                DEMO_TYPE_CAR,
                G_SIGNAL_RUN_FIRST | G_SIGNAL_DETAILED,
                G_STRUCT_OFFSET(DemoCarClass, driven),
                NULL, NULL,
                g_cclosure_marshal_VOID__ULONG,
                G_TYPE_NONE,
                1,
                G_TYPE_ULONG
        );


        g_object_class_install_property(
                gobject_class,
                PROP_MILAGE,
                g_param_spec_ulong(
                        "milage",
                        "milage",
                        "<no␣desc>",
                        0,
                        256,
                        0,
                        G_PARAM_READABLE | G_PARAM_WRITABLE
                )
        );

        g_object_class_install_property(
                gobject_class,
                PROP_NAME,
                g_param_spec_string(
                        "name",
                        "name",
                        "<no␣desc>",
                        "",
                        G_PARAM_READABLE | G_PARAM_WRITABLE
                )
        );


}

GType
demo_car_get_type() {
        static GType this_type = 0;

        if (!this_type) {
                static const GTypeInfo this_info = {
                        sizeof (DemoCarClass),
                        NULL,
                        NULL,
                        (GClassInitFunc) demo_car_class_init,
                        NULL,
                        NULL,
                        sizeof(DemoCar),
                        0,
                        (GInstanceInitFunc) demo_car_init,
                };
                this_type = g_type_register_static(
                                G_TYPE_OBJECT ,
                                "DemoCar",
                                &this_info,
                                0
                );
        }
        return this_type;
}

DemoCar*
demo_car_new() {
        return DEMO_CAR(g_object_new(DEMO_TYPE_CAR, NULL));
}

```

11.2 Example: interface "Demo::Talk"

11.2.1 Header file

```

#ifndef INC_DEMOTALK_H
#define INC_DEMOTALK_H

#include <glib-object.h>

G_BEGIN_DECLS

#define DEMO_TYPE_TALK \
        (demo_talk_get_type ())
#define DEMO_TALK(obj) \
        (G_TYPE_CHECK_INSTANCE_CAST ((obj), DEMO_TYPE_TALK, DemoTalk))
#define DEMO_IS_TALK(obj) \
        (G_TYPE_CHECK_INSTANCE_TYPE ((obj), DEMO_TYPE_TALK))
#define DEMO_TALK_GET_IFACE(obj) \
        (G_TYPE_INSTANCE_GET_INTERFACE ((obj), DEMO_TYPE_TALK, DemoTalkIFace))

typedef struct _DemoTalk DemoTalk;
typedef struct _DemoTalkIFace DemoTalkIFace;

struct _DemoTalkIFace {
        GTypeInterface g_iface;

        /*<vtable>*/
        void (*talk) (DemoTalk *);
};

GType           demo_talk_get_type();

inline void     demo_talk_talk(DemoTalk*);

G_END_DECLS

#endif
```

11.2.2 Source file

```
#include"demotalk.h"

inline void
demo_talk_talk(DemoTalk* self) {
        DEMO_TALK_GET_IFACE(self)->talk(self);
}


GType demo_talk_get_type () {
        static GType this_type = 0;

        if (! this_type) {
                static const GTypeInfo this_info =
                {
                  sizeof (DemoTalkIFace),
                  NULL,
                  NULL,
                  NULL,
                  NULL,
                  NULL,
                  0,
                  0,
                  NULL
                };

                this_type = g_type_register_static (G_TYPE_INTERFACE,
                        "DemoTalk",
                        &this_info,
                        0
                );
                g_type_interface_add_prerequisite (this_type,
                        G_TYPE_OBJECT
                );
        }

        return this_type;
}
```

11.3 Example: class ”Demo::Person”

This class implements the interface ”Demo::Talk” which provide a ”talk” method.

11.3.1 Header file

```
#ifndef INC_DEMOPERSON_H
#define INC_DEMOPERSON_H

#include<glib-object.h>

```

```
G_BEGIN_DECLS

#define DEMO_TYPE_PERSON \
        demo_person_get_type()
#define DEMO_PERSON(obj) \
        (G_TYPE_CHECK_INSTANCE_CAST((obj), DEMO_TYPE_PERSON, DemoPerson))
#define DEMO_PERSON_CLASS(klass) \
        (G_TYPE_CHECK_CLASS_CAST((klass), DEMO_TYPE_PERSON, DemoPersonClass))
#define DEMO_IS_PERSON(obj) \
        (G_TYPE_CHECK_INSTANCE_TYPE((obj), DEMO_TYPE_PERSON))
#define DEMO_PERSON_GET_CLASS(o) \
        (G_TYPE_INSTANCE_GET_CLASS ((o), DEMO_TYPE_PERSON, DemoPersonClass))

typedef struct _DemoPerson DemoPerson;
typedef struct _DemoPersonClass DemoPersonClass;

struct _DemoPerson {
        GObject parent;

        /*< public >*/
        /*< private >*/
        gchar *_name;
};

struct _DemoPersonClass {
        GObjectClass parent;

        /*< public >*/
        /*< private >*/
        /*< signals >*/
        /*< vtable >*/
};

GType           demo_person_get_type();
DemoPerson*     demo_person_new();
void            demo_person_set_name(DemoPerson *self, gchar *name);
gchar*          demo_person_get_name(DemoPerson *self);


G_END_DECLS

#endif
```

11.3.2 Source file

```

#include "demoperson.h"
#include "demotalk.h"

static GObjectClass *parent_class = NULL;


void
demo_person_set_name(DemoPerson *self, gchar *name) {
        self->_name = g_strdup(name);
}

gchar*
demo_person_get_name(DemoPerson *self) {
        return self->_name;
}

static void
demo_person_impl_talk(DemoTalk *self) {
        g_print("Hi,␣I'm␣%s\n", DEMO_PERSON(self)->_name);
}

static void
demo_person_impl_dispose(DemoPerson *self) {
        G_OBJECT_CLASS(parent_class)->dispose((GObject*) self);
}


static void
demo_person_impl_finalize(DemoPerson *self) {
        g_free(self->_name);

        G_OBJECT_CLASS(parent_class)->finalize((GObject*) self);
}

static void demo_person_demo_talk_init (DemoTalkIFace *iface) {
        iface->talk = demo_person_impl_talk;
}

static void
demo_person_init(DemoPerson *self) {
        self->_name = "Anonymous";
}

static void
demo_person_class_init(DemoPersonClass *klass) {
        GObjectClass *gobject_class = G_OBJECT_CLASS(klass);

        parent_class = g_type_class_peek(G_TYPE_OBJECT);

        gobject_class->dispose = demo_person_impl_dispose;
        gobject_class->finalize = demo_person_impl_finalize;

}

GType
demo_person_get_type() {
        static GType this_type = 0;

        if (!this_type) {
                static const GTypeInfo this_info = {
                        sizeof (DemoPersonClass),
                        NULL,
                        NULL,
                        (GClassInitFunc) demo_person_class_init,
                        NULL,
                        NULL,
                        sizeof(DemoPerson),
                        0,
                        (GInstanceInitFunc) demo_person_init,
                };
                this_type = g_type_register_static(G_TYPE_OBJECT ,
                                "DemoPerson",
                                &this_info,
                                0
                );

                static const GInterfaceInfo demo_talk_info = {
                        (GInterfaceInitFunc) demo_person_demo_talk_init,
                        NULL,
                        NULL
                };
                g_type_add_interface_static(this_type,
                                DEMO_TYPE_TALK,
                                &demo_talk_info
                );
        }
        return this_type;
}

DemoPerson*
demo_person_new() {
        return DEMO_PERSON(g_object_new(DEMO_TYPE_PERSON, NULL));
}

```

11.3.3 Main file

```

#include "demoperson.h"
#include "demotalk.h"

gint
main(gint argc, gchar *argv[]) {
        g_type_init();

        DemoPerson *person = demo_person_new();
        demo_person_set_name(person, "Joe");

        demo_talk_talk(DEMO_TALK(person));

        g_object_unref(G_OBJECT(person));

        return 0;
}
```

11.4 Empty class stub

11.4.1 C++

```

#ifndef INC_TESTEMPTYOBJECT_H
#define INC_TESTEMPTYOBJECT_H

#include <gobject>

namespace Test {
        class EmptyObject : public G::Object {
                public:
                        EmptyObject();
                        ~EmptyObject();
        };
};

#endif
```

```

#include "testemptyobject.h"

Test::EmptyObject() {

}

Test::~EmptyObject() {

}
```

11.4.2 Glib-C

```

#ifndef INC_TESTEMPTYOBJECT_H
#define INC_TESTEMPTYOBJECT_H

#include<glib-object.h>

G_BEGIN_DECLS

#define TEST_TYPE_EMPTY_OBJECT  \
test_empty_object_get_type()
#define TEST_EMPTY_OBJECT(obj)          \
(G_TYPE_CHECK_INSTANCE_CAST((obj), TEST_TYPE_EMPTY_OBJECT, TestEmptyObject))
#define TEST_EMPTY_OBJECT_CLASS(klass)  \
(G_TYPE_CHECK_CLASS_CAST((klass), TEST_TYPE_EMPTY_OBJECT, TestEmptyObjectClass))
#define TEST_IS_EMPTY_OBJECT(obj)       \
G_TYPE_CHECK_INSTANCE_TYPE((obj), TEST_TYPE_EMPTY_OBJECT))
#define TEST_EMPTY_OBJECT_GET_CLASS(o)  \
(G_TYPE_INSTANCE_GET_CLASS ((o), TEST_TYPE_EMPTY_OBJECT, TestEmptyObjectClass))

typedef struct _TestEmptyObject TestEmptyObject;
typedef struct _TestEmptyObjectClass TestEmptyObjectClass;

struct _TestEmptyObject {
        GObject parent;

};

struct _TestEmptyObjectClass {
        GObjectClass parent;

};

GType                   test_empty_object_get_type();
TestEmptyObject*        test_empty_object_new();

```

```
G_END_DECLS

#endif
```

```

#include "testemptyobject.h"

static GObjectClass *parent_class = NULL;

static void
test_empty_object_impl_dispose(TestEmptyObject *self) {
        G_OBJECT_CLASS(parent_class)->dispose((GObject*) self);
}

static void
test_empty_object_impl_finalize(TestEmptyObject *self) {
        G_OBJECT_CLASS(parent_class)->finalize((GObject*) self);
}

static void
test_empty_object_init(TestEmptyObject *self) {

}

static void
test_empty_object_class_init(TestEmptyObjectClass *klass) {
        GObjectClass *gobject_class = G_OBJECT_CLASS(klass);

        parent_class = g_type_class_peek(G_TYPE_OBJECT);

        gobject_class->dispose = test_empty_object_impl_dispose;
        gobject_class->finalize = test_empty_object_impl_finalize;
}

GType
test_empty_object_get_type() {
        static GType this_type = 0;

        if (!this_type) {
                static const GTypeInfo this_info = {
                        sizeof (TestEmptyObjectClass),
                        NULL,
                        NULL,
                        (GClassInitFunc) test_emptyobject_class_init,
                        NULL,
                        NULL,
                        sizeof(TestEmptyObject),
                        0,
                        (GInstanceInitFunc) test_emptyobject_init,
                };
                this_type = g_type_register_static(
                                                G_TYPE_OBJECT ,
                                                "TestEmptyObject",
                                                &this_info,
                                                0
                );
        }
        return this_type;
}
```

```

TestEmptyObject*
test_empty_object_new() {
        return TEST_EMPTY_OBJECT(g_object_new(TEST_TYPE_EMPTY_OBJECT, NULL));
}
```

Bibliography

[Ber91] P.L. Bergstein, editor. *Object Preserving Class Transformations*, 1991.

[Boo91] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings series in Ada and software engineering. Benjamin/Cummings, 1991.

[Bud97] T. Budd. *An Introduction to Object Oriented Programming.* Addison Wesley, 2nd edition, 1997.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *In Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Internation Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[DDN03] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufman Publishers, 2003.

[E.91] Casais E. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* PhD thesis, University of Geneva, 1991.

[Fow00] Martin Fowler. *Refactoring, Improving the design of Existing Code.* Addison-Wesley, 2000.

[Gar] Alejandra Garrido. Software refactoring applied to C programming language. Master's thesis, Universidad National de la Plata.

[GJ03] Alejandra Garrido and Ralph Johnson. Refactoring C with conditional compilation. In *IEEE International Conference on Automated Software Engineering*, number 8th, Montreal, Canada, 2003.

[GSMD03] Pieter Van Gorp, Hans Stenten, Toms Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. Technical report, University of Antwerp, Belgium - Université de Mons-Hainaut, Belgium, 2003.

[IL90] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. Technical report, Stanford University, Computer Systems Laboratory, March 1990.

[Jr.87] Frederick P. Brooks Jr. No silver bullet essence and accidents of software engineering. *IEEEComputer*, pages 10–19, April 1987.

[JR97] Roberts D. Brant J. and Johnson R. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[Lip96] Stanley B. Lippman. *Inside the C++ Object Model.* Addison Wesley, 1996.

[Mad88] Ole Lehrmann Madsen. What object-oriented programming may be - and what it does not have to be. *Lecture Notes in Computer Science*, 1988.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, 2nd edition, 1997.

[MIKC92] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in C++: an operating systems perspective. Technical Report UIUCDCS–R–92–1736, Department of Computer Science, Urbana-Champaign, 1992. url : http:///citeseer.ist.psu.edu/madany92reification.html.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois, 1992.

[PJ00] Meiler Page-Jones. *Fundamentels of Object-Oriented Design in UML.* Addison Wesley, 2000.

[Pre97] Roger S. Pressman. *Software Engineering: A practitioner's approach.* McGraw-Hill, 4th edition, 1997.

[RBJ97] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 1997.

[Rob99] Don. Roberts. *Eliminating Analysis in Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[Som96] Ian Sommerville. *Software Engineering.* Addison Wesley, 5th edition, 1996.

[Str91] Bjarne Stroustrup. What is "object-oriented programming"? In G. Goos and J. Hartmanis, editors, *Proc. European Conf. on Object-Oriented Programming*, Paris (France), 1987, revisited 1991. Springer-Verlag, Lecture Notes in Computer Science no. 276.