
THESIS

A. Endrodi

2007

University of Pannonia, Hungary
Department of Information Systems
Information Technology Degree Program

THESIS

Evolution of MenuGene A Dietary Counselling System

A. Endrodi

Supervisor: Gaál, Balázs

2007



PANNON EGYETEM
MŰSZAKI INFORMATIKAI KAR
Információs Rendszerek Tanszék

Veszprém, 2006. október

DIPLOMATÉMA-KIÍRÁS

Endrődi Ádám

műszaki informatika szakos hallgató részére

A MenuGene táplálkozás-tanácsadó szakértői rendszer evolúciója

Témavezető: Gaál Balázs

A feladat leírása:

Olyan szoftverrendszer létrehozása, amely a fogyasztó speciális igényeire tekintettel képes számára professzionális étrendeket előállítani, számos szakember munkáját igényli, és műszaki problémák garmadáját veti fel. A legnagyobb akadály, hogy nincs bevett algoritmizált módszer az étrendek összeállítására, továbbá olyan, jól használható szoftveres infrastruktúra sem létezik, amely támogatná a táplálkozás-tanácsadó szakértői logika megalkotását.

A jelölt feladata, hogy részt vegyen a Pannon Egyetem Információs Rendszerek Tanszéke és a Semmelweis Egyetem Dietetikai Tanszéke által közösen fejlesztett MenuGene táplálkozási szakértői rendszer tervezési, fejlesztési és kialakítási munkálataiban.

Részletesen:

- Olyan keretrendszert hozzon létre, amely a fejlesztői kívánalmaknak jól megfelel, és olajozottan együttműködik egy előre meghatározott genetikus algoritmus függvénykönyvtárral.
- Az integráció és a működés közben (pl. adatbázis illesztéskor) fellépő problémák leküzdéséhez szükség szerint írjon segédprogramokat.
- Járuljon hozzá a MenuGene rendszerstruktúrájának alakításához minden tervezési szinten.
- Különös tekintettel ügyeljen a megvalósítás karbantarthatóságára, helyességére, robusztusságára és szervizelhetőségére.
- Tegye a rendszer szolgáltatásait egyszerű formában elérhetővé a számítástechnikában kevésbé járatos emberek számára is.
- Készítse fel a rendszert a végfelhasználói igénybevételre, értékelhető funkcionalitás kielégítő módon történő közvetítésére.

Záróvizsga tárgycsoportok:

- Kommunikációs rendszerek:
Internet és TCP/IP, Kommunikációs protokollok és számítógép-hálózatok tervezése
- Számítógép hálózatok és hírközlésmélet:
Számítógép hálózatok, Információ és hírközlésmélet
- Egészségügyi információs rendszerek:
Egészségügyi információs rendszerek I, Számítógépes döntéstámogatás

Gaál Balázs
témavezető

Dr. Kozmann György
tanszékvezető

Nyilatkozat

Alulírott Endrődi Ádám diplomázó hallgató, kijelentem, hogy a diplomadolgozatot a Pannon Egyetem Információs Rendszerek Tanszékén készítettem mérnök-informatikus diploma (master of engineering in information technology) megszerzése érdekében.

Kijelentem, hogy a diplomadolgozatban foglaltak saját munkám eredményei, és csak a megadott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem azt, hogy a diplomadolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

May 21, 2007

.....
Endrődi Ádám

ABSTRACT

This thesis is the technical and historical account of the MenuGene project, conducted by the Department of Information Systems and developed in joint partnership with the Department of Dietetics of the Semmelweis University, Budapest, which the author has participated in as a software developer. The project goal is to establish a computer service for the layperson to help in the compilation of personal dietary menus that offer optimal nourishment and satisfy the consumer's taste as well.

The system is composed of several layers and components: standalone user interfaces and a web interface for user interaction, a database for static nutrient data storage, and a system service that solves the problem of menu generation per user request, all linked through network. The problem solver logic is based on genetic algorithms.

Mature or experimental technologies exist for all of these components. The completed objective of the project was to fit them together in appropriate implementations, and to create a convenient research platform for the further improvement of the problem solver engine, taking special note of the future maintainability of the product.

To this end several programs, applications, libraries and developer utilities have been written in Perl, C, C++ and PHP languages, and the inherited database schema has been reworked. The source code is equipped with excessive documentation, which the present thesis is also part of. As to the current state of affairs, the system is demonstratable.

This document covers the software engineering aspects of MenuGene. It reviews the tools and practices that characterized the development process, along with the implementation technologies and methods, aiming at giving sufficient explanations concerning decisions for or against a particular item. A detailed overview of genetic algorithms is also included.

Keywords: MenuGene, nourishment, genetic algorithms, multi-objective optimization, software engineering, defensive programming

TARTALMI ÖSSZEFOGLALÓ

Jelen dolgozat a MenuGene projekt műszaki és történeti háttéréről számol be. A projektet az Információs Rendszerek Tanszékének égisze alatt, a Semmelweis Egyetem Dietetikai Tanszékével közösen mozdították előre, amelyben a szerző szoftverfejlesztőként működött közre. A program célja egy olyan informatikai szolgáltatás megalapozása, amely képes az átlagembereknek hatékony segítséget nyújtani étlapjuk összeállításában úgy, hogy az mind táplálkozási szükségleteiknek, mind egyéni ízlésüknek maximálisan megfeleljen.

A rendszer részei között akadnak önállóan, saját számítógépen futtatható felhasználói programok, web alapú felület, időben nem változó tápanyag-információkat tároló adatbázis, valamint egy háttérben futó rendszerprogram, ami felhasználói kérésre megfelelő menüt állít elő. Mindezen részek hálózaton keresztül tartják egymással a kapcsolatot. A rendszer a megfelelő menüt genetikus algoritmusok segítségével keresi meg.

A felsorolt komponensek létrejöttéhez rendelkezésre állnak kiforrott, vagy legalább kísérleti fázisban álló technológiák. A dolgozat mögött meghúzódó munka célja az volt, hogy ezeket a technológiákat alkalmas megvalósításokba bújtatva összeálljon a rendszer, továbbá hogy kiépüljön egy jól használható keretrendszer az alacsony szintű problémamegoldó módszer további finomításához. A munkálatok során a karbantarthatóság különösen lényeges szerephez jutott.

A feladat elvégzéséhez számos programot, felhasználói alkalmazást, programkönyvtárat és segédprogramot kellett kifejleszteni Perl, C, C++ és PHP nyelveken; ezen kívül az adatbázis már meglévő sémája is alapos átdolgozáson esett át. A forráskódbázis jelentős mennyiségű dokumentációval rendelkezik, amelyet a jelen dolgozat is gyarapítani szándékozik. A rendszer készültségi állapota elérte a bemutatathatóság határát.

Az elkövetkezendő írás a MenuGene mérnöki szoftverfejlesztési oldalát mutatja be. Áttekintésre kerülnek a fejlesztési folyamat során alkalmazott eszközök és konvenciók, csakúgy mint a különböző megvalósítási technológiák és módszerek. A szöveg erős hangsúlyt fektet a folyamat során meghozott döntések alátámasztására, valamint helyet kapott a genetikus algoritmusok általános ismertetése is.

Kulcsszavak: MenuGene, táplálkozás, genetikus algoritmusok, többszempontú optimalizálás, hosszútávú szoftverfejlesztés, védekező programozás

Contents

1	Executive summary	6
1.1	Purpose	6
1.2	Structure	7
1.3	The works	11
2	Overview of the implementation	15
2.1	Brief history	15
2.2	Qualitative goals	17
2.2.1	Maintainability	19
2.2.2	Correctness	20
2.2.3	Robustness	22
2.2.4	Serviceability	23
2.2.5	Care	24
2.3	Development environment	25
2.3.1	Platform	25
2.3.2	Programming languages	25
2.3.3	Compilers	26
2.3.4	Build system	26
2.3.5	Software configuration managers	26
2.3.6	Developer tools	27
2.3.7	Database managers	27
2.3.8	Documentation	28
2.3.9	Quality assurance tools	28
2.4	The codebase	29
2.4.1	A walk through the repository	29
2.4.2	Dependencies	33
2.4.3	How to build it	34
3	Implementation details	37
3.1	Object model	38
3.1.1	Basic facilities	38
3.1.2	Copy on write	42
3.1.3	Containers	43
3.2	Logging	46
3.3	Error handling	49
3.4	Collections	51

CONTENTS

3.5	XML driver	57
3.6	Test suite	60
4	In closing	62
5	Appendices	63
5.1	Bibliography	63
5.2	Progliography	65
5.3	Supplementary figures	68
5.4	Screenshots	70
5.5	Annotated database schema	72

List of Figures

1.1	Information flow and levels of application of MenuGene	7
1.2	Example deployment configurations of MenuGene	8
1.3	System structure	10
1.4	Entities involved with the genetic algorithm	12
1.5	Internal representation of menu hierarchies	13
3.1	The effect of dup()	40
3.2	The process of copying on write	44
3.3	The structure of gsmon	47
3.4	Internal representation of a GSavl collection	52
3.5	Internal GSxml representation of an XML document	58
5.1	The fitness function of MenuGene	68
5.2	Possible relations between solutions and attributes	69
5.3	Screenshots of junkie	70
5.4	Screenshot of subborn	71
5.5	Screenshot of rulez	71

How to read this document

This text aims at documenting the development process of MenuGene as perceived by the author, in the inarticulate hope of attracting successors who would carry on the work. In this regard it can be read as a manual, which helps in finding the way around while making improvements to the system. Those who have this intention will definitely learn most from the **third chapter**. The ones who do not wish to dwell into technical details, but are interested in the *whats* are best served by the **first chapter**. Finally, those professionals who are not very excited by the high-level mission of the system, nor by its implementation nuts and bolts may find the **second chapter** a good reading into software engineering.

The scope of this document is limited to those bodies of knowledge that are not apparent from the source code. This is to avoid overlaps, which would inevitably reduce the usefulness of this text. It follows that we abstained from including lengthy code samples. For this the source codebase should be consulted, which we tried to make as seamless as possible by the means of crossreferences.

annotation

The text is extensively annotated and crossreferenced to foster the rapid overview of sections and the quick localization of additional information. In the electronic version nearly all paragraphs are bookmarked, which gives the impression of a very detailed table of contents if displayed with a suitable viewer. This case the crossreferences behave as hyperlinks. In the hardcopy edition this is emulated by indicating the referred page number on the margin, out of the text body to minimize distraction in reading.

Crossreferences are colored differently, depending on the type of their destinations. **Internal links** take the reader to another point within the text body, including tables and figures. **Citations** point to entries in the **bibliography** or in the **progliography** (list of referred software). **External references** are meant to help in looking up the authentic source of information, which can be checked up. Unless it is clearly visible otherwise these links point to files and directories in the source codebase. To leverage the immediate access offered by them the repository must be extracted in `/src`.

Many words in the text are set in distinctive typefaces to facilitate the semantic decoding of dense technical material. Code fragments and path elements are set in **typewriter** style, allowing for the unambiguous identification of functions and variables. Proper technical names are **sans serif**, while SMALL CAPS are reserved for concrete, widely recognized software names cited from the progliography. *Italics* emphasize key points in a phrase, or markup auxiliary subjects.

Occasionally a little sign¹ demands attention on the margin, warning that the following topic might be hard to swallow. Its rough interpretation is that we suggest that readers who cannot make sense of the paragraph in question at the first sight just ignore the explanations and accept the statements as they are.



¹Borrowed from [11], whose author placed it into public domain.

colophon This document was typeset by [TEX](#), employing hundreds of custom macros organized into an integrated package developed by the author while writing this thesis. Most figures have been prepared with [GRAPHVIZ](#), its PostScript output being heavily customized by other homegrown tools. In this work [[1](#), [4](#)] have proven indispensable. During composition the safety and the consistency of the document sources were guarded by the [DARCS](#) revision control system, which turned out to be extremely suited for this kind of situations.

Chapter 1

Executive summary

1.1 Purpose

MenuGene is a system of software for computer aided dietary counselling. Its ultimate goal is to help ordinary people in the assembly of daily and weekly dietary menus—what to have for breakfast, lunch etc. so that their nutritional needs, dietary allowances and taste are all satisfied.

From the system’s point of view a person’s *nutritional needs* are defined by the optimal daily consumption of elementary ingredients, such as carbohydrate, protein, and lipids. *Dietary allowances* restrict the range of dishes consumable by the subject, as in the case of people with milk susceptibility. *Taste* expresses preferences towards or against particular kinds of meals, or any combinations thereof.

intended audience

The intended audience extends from the ill to healthcare providers. *People with diabetic disabilities* benefit in breaking the routine meals by discovering new recipes that are as fitting to their special needs as the ones prescribed by dietary consultants. *Healthy individuals* conscious about their condition will find it a cheap and easy way to improve their nutrition. *Domestic partners* are relieved of deciding what to prepare for their families, so that everyone’s taste is respected. *Dietary experts* may be able to give their clients better advice. Finally, *healthcare providers* may find it appealing as a low-cost addition to their existing portfolio.

use cases

MenuGene has three levels of application. The mid-level usage enables consumers to store, represent and communicate dietary information *structurally*: recipes, nutritional needs, dietary allowances, taste. The high-level use case implies the previous one, and in addition the system constructs (*generates*) a menu plan by artificial intelligence methods, attempting to observe all the constraints expressed with regards to an ideal menu. On the lowest level MenuGene is *designed* to be a platform for research in these fields of information and knowledge representation, and artificial intelligence.

deployment configurations

MenuGene can be deployed and accessed several ways, as illustrated in **figure 1.2**. In the simplest case (1.2(a)) the consumer is in direct interaction with the system, telling about ^{her}_{his} nutritional needs (perhaps deriving them from popular literature), dietary allowances, and preferences, then interpreting the

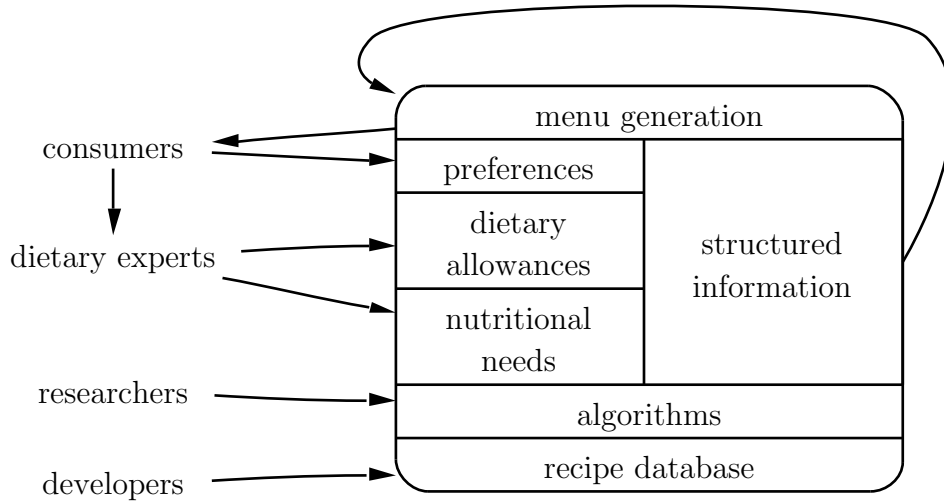


Figure 1.1: Information flow and levels of application of MenuGene

auto-generated menu plan without external assistance. This setting is well-suited for voluntary and/or regular uses of the service, when the fitting of the menu is not critical or the consumer has already gathered enough experience with it. It is also possible to draw on a third-party consultant (1.2(b)) estimating the personal requirements and formulating professional advice based on the machine-generated plans. This case the consumer may not even be aware of the working of the computer aid, which is convenient for those who do not wish to engage in technical details. The configuration depicted in figure 1.2(c) is a synthesis of the former ones: consumers are given freedom in specify their preferences, while the responsibility of determining dietary requirements remains on the shoulders of experts. Finally (1.2(d)), the system can be envisaged as an integrated part of another comprehensive advisory system, such as CORDELIA. This setting is best for healthcare providers whose existing services can gain from the structured personal dietary information held by MenuGene.

The presence of ‘developers’ in the figures is meant to emphasize the importance of the continuous maintenance and enhancement of the MenuGene databases, without which the system would quickly degrade into obsolescence.

1.2 Structure

The system consists of several components which we can categorize roughly into three groups, layered upon the top of one another. (figure 1.3(a)) The first layer interacts with the *end-user*, accepting requests and presenting results in human-friendly fashion. The decoded queries are passed onto the component that has a global view on the problem to be solved and *controls* its resolution, which is done at the bottom by specialized, reusable *libraries*. The communication within the system and toward end-users is mostly networked, which

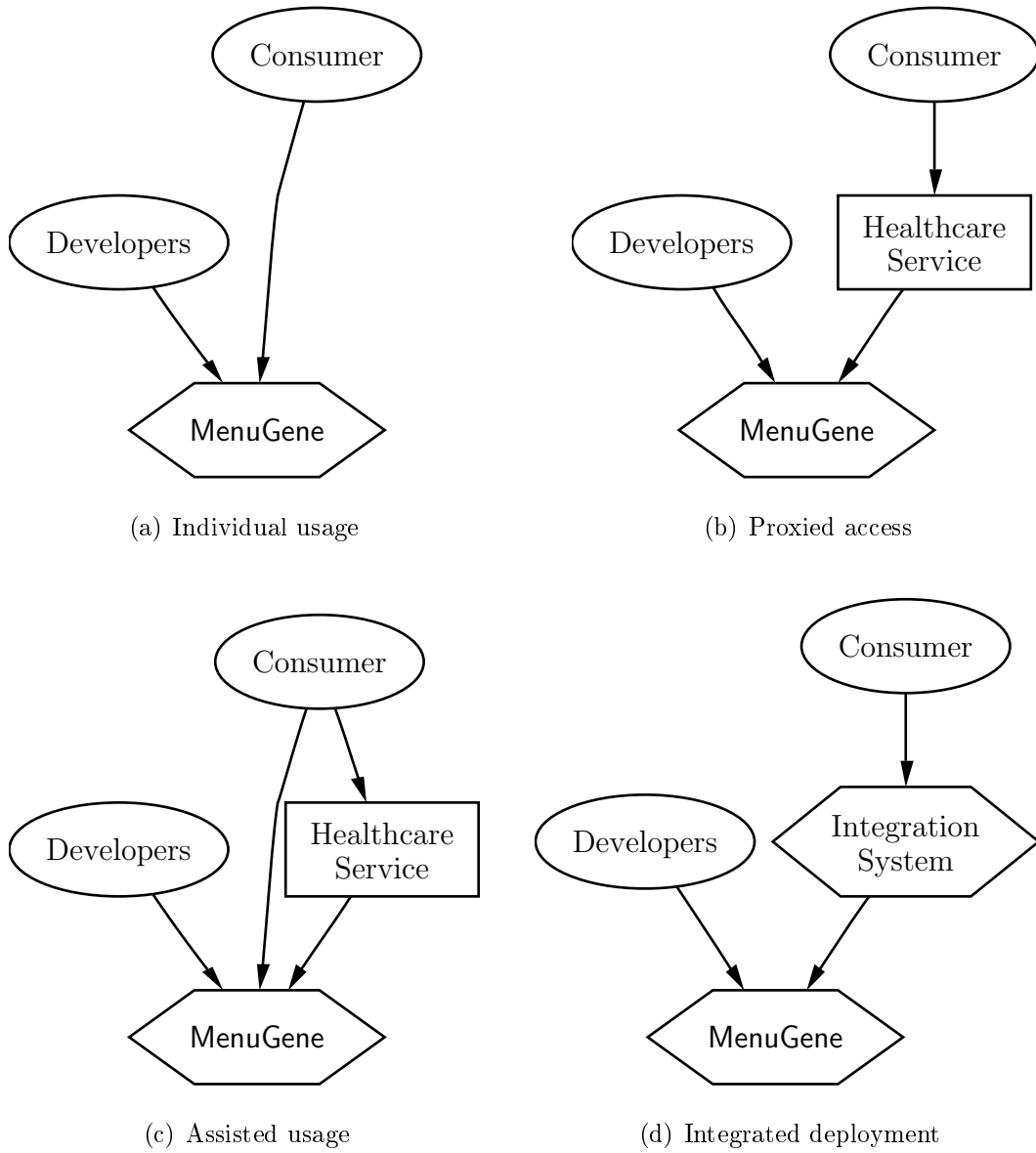


Figure 1.2: Example deployment configurations of MenuGene

allows more scalability and flexibility in deployment. [Figure 1.3\(b\)](#) shows the *use*-relationships between the actors, whose detailed description follows.

junkie *Consumers*, who need dietary advice are served by **junkie**, the web front-end of MenuGene. Users can enter their nutritional needs into HTML forms, the program relays this information to the controller, and displays the tabulated daily or weekly menu recommendation along with goodness estimations when the response is ready. The interface is multi-lingual (currently English and Hungarian are supported) and can be embedded into other web services, making it possible to take user requirements from sources other than its own forms. [Figure 5.3](#) shows screenshots of the interface.

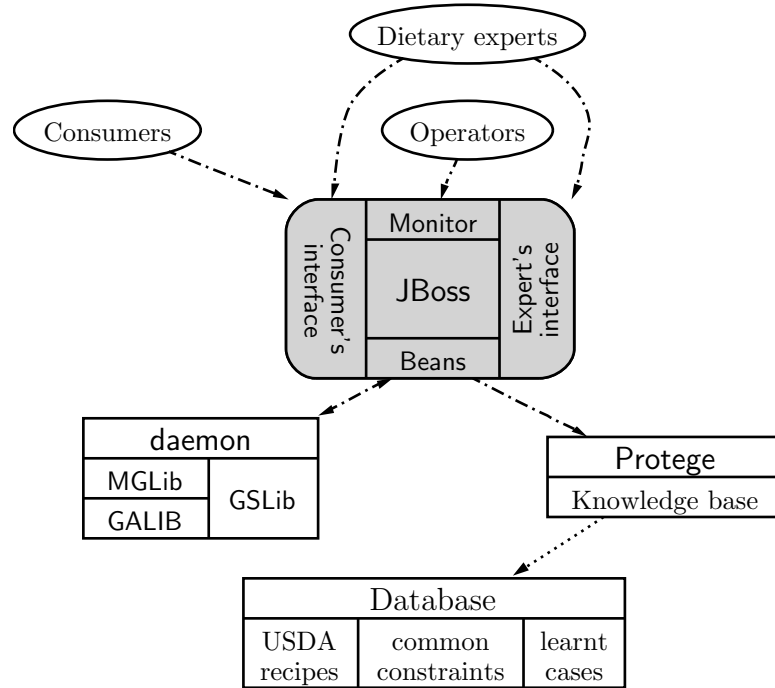
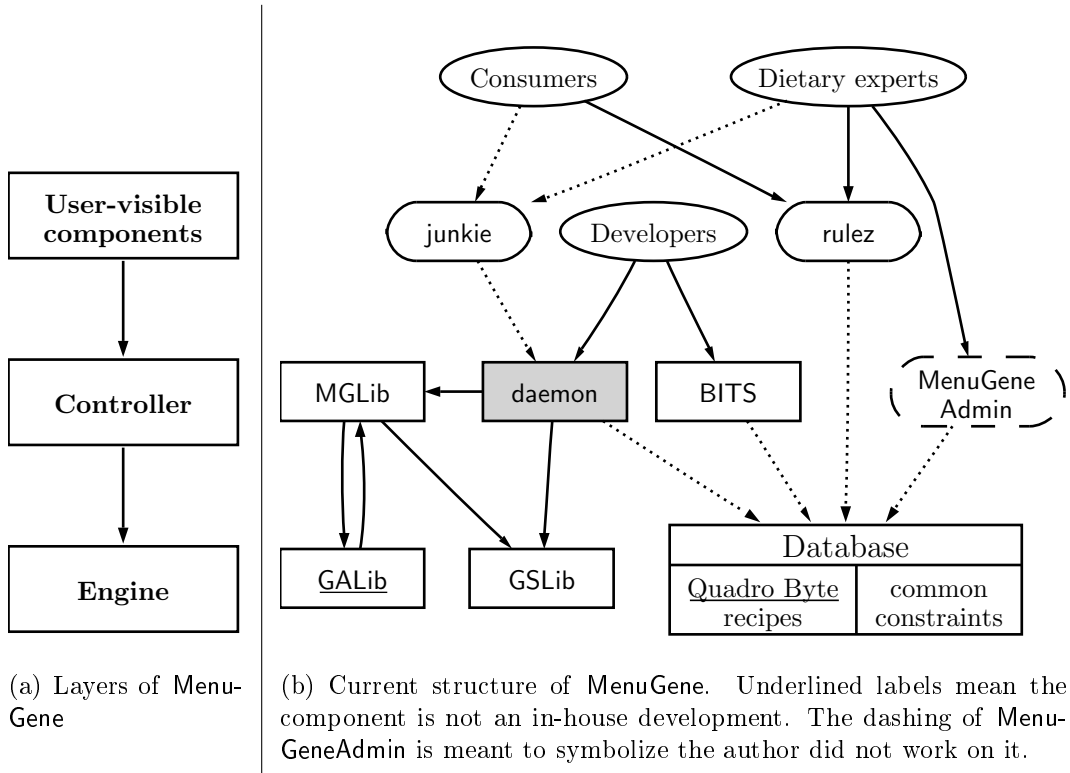
MenuGeneAdmin The system's another user-accessible entry point is **MenuGeneAdmin**, which is intended for *dietary experts*. This application runs on the local computer and allows editing the MenuGene database remotely through a graphical user interface.¹ The database supply the controller with bulk, static information essential for the menu generation. It is static in the sense that its contents are not subject to change in the course of problem solving. The database is loaded with recipes (acquired commercially from [QUADRO BYTE](#)) that the final menu recommendation will be consist of, groupings of the recipes (such as puddings, soft drinks, or fast food), and a set of constraints applicable in typical circumstances (i.e. to people suffering from some common disease). Privileged dietary experts have means to improve and customize this data set by **MenuGeneAdmin** in the interest of their clientele.

rulez The last user-visible application of the system is **rulez**, which can be used by all end-users to edit **rules**. The integration of this program is still in the alpha stage.

daemon The heart of MenuGene is the **daemon**, which formulates the problem in exact terms based on the information coming from **junkie** and the static data fetched from the database. (*“Given ... find ... such that ...”*) Then it configures the engine and calls upon for problem resolution. Depending on the configuration this can be a rather time-consuming process. Whilst it is proceeding run-time status information about the engine is available from the (kind of barefoot) **monitoring facility** of the **daemon**. Besides the regular network interface used by **junkie** the program can be operated from the command line, and also is accessible via CORBA [\[15\]](#) (not merged into mainline). The former is reserved for developers, the latter is held back until future experimentation.

engine Once the engine starts it works unsupervised until a satisfactory solution is found. Menus are generated by **genetic algorithms** (GAs) in collaboration between two software libraries. [GALIB](#) is an external library (with some local adjustments) implementing generic GAs, which call back into [MGLib](#) for problem-specific algorithmic bits. The work of both [MGLib](#) and **daemon** is assisted by [GSLib](#), which essentially defines a convenient general framework for *clean* software development. None of the libraries has any knowledge about the database and its internal schemata.

¹The claims no affiliation with the creation of this software, and takes no responsibility for it.



(c) A possible architecture for the enterprise market.

Figure 1.3: System structure. Arrows express *use*-relationship and direction. The line style indicates the kind of communication transport. Solid lines: local, dotted lines: (inter)networked, dot-dash: RMI/ CORBA. Nodes with rounded corners are visible to end-users. Box shading emphasizes the centrality of a component.

BITS **BITS** of software play an important role in the life of MenuGene developers. It is a collection of **miscellany utilities** which take no part in the normal operation of the system, but are supportive of the development process. They include debug aids, visualizers, database maintenance tools and more.

future **Figure 1.3(c)** shows a substantially different system architecture envisaged for the enterprise in admission that that market demands more scalability than MenuGene offers currently. In the proposition **daemon** should lose its central position, taken over by a **JBoss** conglomerate in its place. User-visible interfaces would be sited in the application server, giving users the feel of a local-running program by means of dummy proxying mock-ups (not included in the figure). Bulk data would be hidden behind a ‘knowledge base’, which can represent complex relationships between pieces of information that is hard to model in relational databases. With respect to the database, the basic data set of **QUADRO BYTE** is going to be **superseded** by the one of the U.S. Department of Agriculture (USDA [20]), which is believed to be of higher quality. Also, we are planning on making room for dynamic hints derived from past runs for the **daemon**, so it could start off searching for optimal menu plans from points proven to be favorable. This is called *case-based menu generation*. All these areas have been worked on, reaching varying status of readiness.

1.3 The works

In this section we will give a high-level outline of the workings of MenuGene that produces menus satisfying **requirements**. In this text we will only cover the underlying theory to the extent, and from the perspective of that is necessary to facilitate the understanding of the topic in focus. Interested readers should follow up [7].

The engine operates in the *problem space*. This imaginary space is filled with *solutions* to the given problem. In the context of MenuGene a solution is, certainly, an optimal weekly menu plan, while the problem space is the (finite) set of all possible weekly dietary menu plans. Thus, the duty of the engine is to *pick out* a solution from the problem space that matches the consumer’s needs as closely as possible.

entities The engine utilizes *genetic algorithms* (GA) for this purpose. GAs maintain a subset of the problem space called *population*, which restricts what solutions the GA can consider *at a given time*. The size of the population is chosen so that computations over the subset taking linear or higher order time become feasible. Solutions elected into the population are the *genomes*. Which genomes these are is of cardinal importance, since the final solution to the problem will be chosen from them when the algorithm ends.

genetic algorithm It follows that the task of the GA is to bring forth a population that features the genome which is a (near) optimal solution to the problem the MenuGene engine set out to solve. For this the genetic algorithm improves its population incrementally. Iterations of the GA are called *generations*. In each generation the GA *evolves* the current population, changing the constituting genomes in some or other way around. In formal terminology some of the genomes are

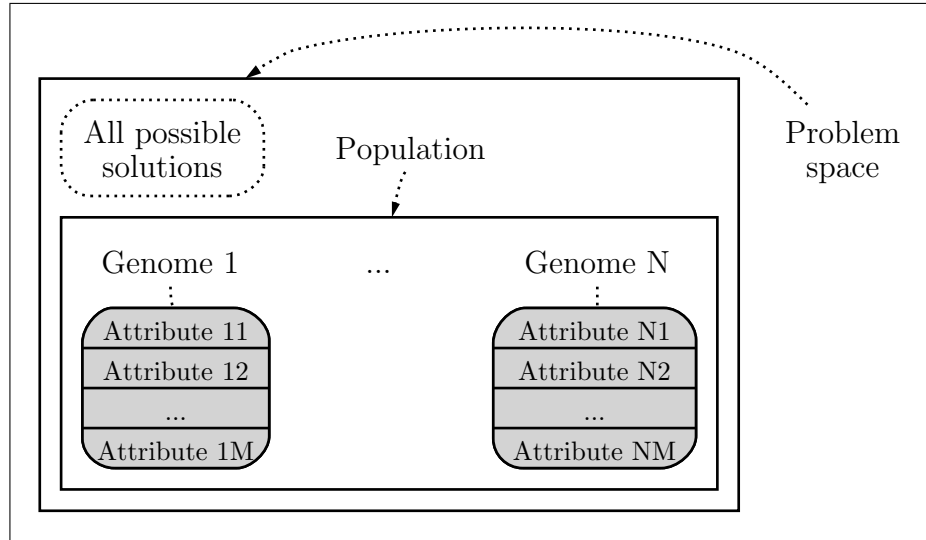


Figure 1.4: Entities involved with the genetic algorithm. Dotted lines link labels to their corresponding drawing.

replaced by solutions from the problem space. This process may be driven by the intention to get closer to a set of genomes that appear to have the desired characteristics, or conversely to back off because the current population seems to be a dead end.¹

Actually, the population is evolved by considering small subsets thereof. This is why a final *selection* is done before the next generation, taking all proposed members of the new population into account at the same time. The algorithm ends when some *terminal condition* is reached. The following pseudo-code summarizes the main points:

```
sub genetic_algorithm(population)
{
    population = selection(evolution(population))
    until terminal_condition(population);
    return best_of(population);
}
```

encoding In order to be of any use genomes must *encode* the solutions they represent in a way the applied genetic algorithm understands, making it possible to evaluate, compare and evolve them. Figure 1.4 has already alluded to a potential encoding, in which genomes are partitioned into several compartments called *attributes* or *genes*. An attribute has a name by which it can be uniquely identified in a given genome. Every attribute encodes a different piece of information. The type of this information (the value of the attribute) is arbitrary (e.g. numbers, text, or any complex data structure). It is worth noting that many genetic algorithms expect that the genomes in the population have the same set of attributes.

¹The special case is when the evolution does not have any particular goal in mind; this is called *random search*.

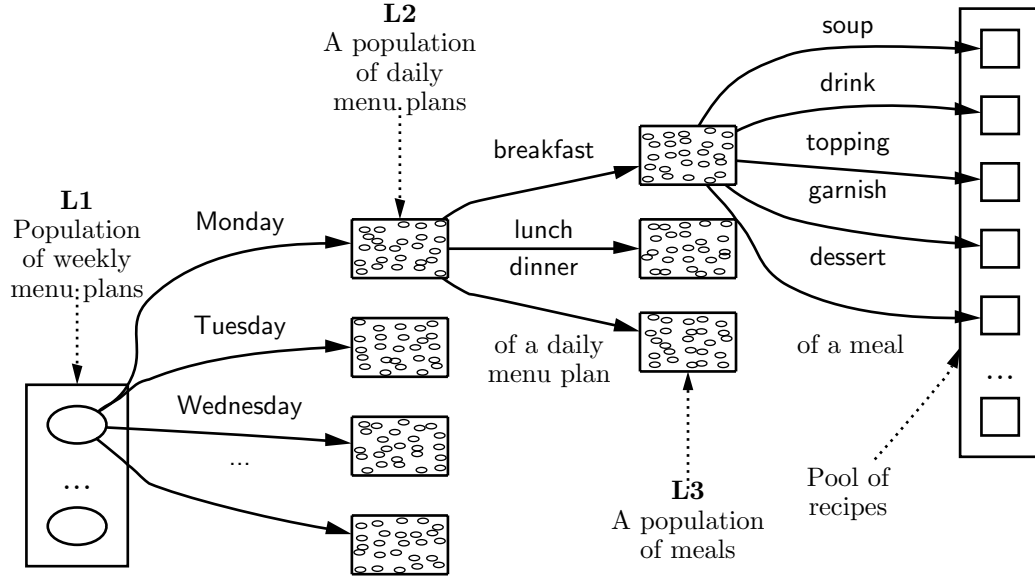


Figure 1.5: Internal representation of menu hierarchies. Ellipses stand for GA genomes, arrows indicate attribute-of relationships between. Note that the arrows originating from the rectangles of populations are meant to signify that all genomes contained have the same set of attributes, only with different values.

The engine of **MenuGene** employs three kind of genomes: weekly menu plans, daily menu plans and meals. Each is bred in separate populations, not allowing for mixins. The relation between the different kinds of genomes is hierarchic, as depicted in [figure 1.5](#) along with the corresponding attributes. The point is that *the values of the attributes of all three levels of nutrition are solutions themselves picked from an appropriate problem space*. This is the distinctive feature, which we shall credit to [7] explicitly, making it possible to break down a complex problem into several smaller ones with fewer dimensions (*divide & conquer*).

genetic operators

The genetic algorithms of **MenuGene** have two means of evolving genomes. One of them is the standard *crossover*, which takes a pair of genomes (the parents) and results in a pair of new ones (the children). All attributes of the parents live on in the children, but it is up to the crossover operator which child inherits a particular attribute. For example, crossing the genomes in [figure 1.4](#) may result in two children having { attribute_{N1}, attribute₁₂, ..., attribute_{NM} } and { attribute₁₁, attribute_{N2}, ..., attribute_{1M} }.

Mutation is the other way (as far as **MenuGene** is concerned) to renew a genome. From the genetic algorithm's point of view by mutation a single genome is replaced by another solution. In **MenuGene** this is implemented by asking some attributes of the genome under mutation to acquire a new value. How the attribute responds is a function of its type, opening the scene for three alternatives.

attribute types

GA-type attributes (L1 and L2 only) run a genetic algorithm (which is private to the attribute), and the new value of the attribute will be the best solution of this GA can produce. POOL-type attributes choose their values

from a preset static pool of genomes randomly. In MenuGene recipes are in a pool, because they do not change during generation. (The system generates menus, not recipes.) A COMPOSIT attribute masters a set of subordinate attributes and asks each of them to renew its value by whatever means it is capable of. At the end COMPOSIT master will choose the best solution presented by its subordinates. This kind of relationship is useful when the problem space has sensible (to humans) divisions, such as coldcut breakfast and brunch.

These mechanisms are stacked recursively (i.e. a COMPOSIT attribute may have COMPOSIT subordinates). Figure 5.2 illustrates all the meaningful combinations of relations between different attribute types. The pool of recipes are read from a database, whose annotated schemata is reproduced in appendix 5.5. The included listing is intended as a starting point to grasp the conception of dietary menus of MenuGene.

evaluation To determine which one is the best, solutions are *evaluated* and compared to each other. Naturally, the evaluation must reflect the consumer's requirements: her nutritional needs, dietary allowances, and taste. The first of these are taken care of by the first pass of the evaluation, the *fitness function*. In MenuGene nutritional needs are expressed by a *constraint set*. For a given nutrient a constraint specifies its expected minimal and maximal amount in a solution, along with the desired optimum.

fitness calculation A solution (weekly, daily menu or a meal) may have multiple constrained nutrients, all of which the fitness function must take into account (*multi-objective optimization*). First it *decodes* the solution to be evaluated, and calculates the cumulative amount of the nutrients in question. These figures are realized at the bottom of the GA hierarchy by genomes from a POOL representing recipes. Each sum is used to sample a curve parametrized by the corresponding constraint; an example is given in figure 5.1. The grand total of these samples constitutes the *fitness score* of the solution.

rules Per se the fitness score provides enough information for the genetic algorithm to estimate the 'goodness' of a solution. Before that, however, the remaining consumer objectives (dietary allowances and taste) are accounted for. Recall that these requirements state what dishes or what combinations thereof the consumer should not/must not/would not eat. MenuGene handles these within its rules subsystem. A rule is essentially a pattern of solutions. If the pattern matches the contents of a solution (e.g. the items of a daily menu) its fitness score is scaled with factor associated with the rule.¹

It is possible to attach constraints and rules at any levels of the GA hierarchy.

Running times To speed up menu generation the engine employs some optimization techniques, notably caching the sums in the fitness calculation, and the copy on write manipulation of genomes. Running times vary depending on the configuration of the genetic algorithms, but it has proven capable of completing the task in five seconds for daily menus, and in half a minute for weekly menus.

¹We have intentionally omitted many details; more information can be found in [src/MGLib/doc](#) of the source repository.

Chapter 2

Overview of the implementation

2.1 Brief history

Whenever someone meets a sizeable piece of software with the intention of dwelling into its internals questions are naturally raised about the events that has influenced its history in order to put its current state into context, allowing for predictions. The following account was partly recovered from memory, and as such, at times it cannot be free from personal tone. Certainly, many episodes had to be omitted, save for the most important ones. Interested readers can learn more from the [commitlogs](#).

zeroth generation

MenuGene started its existence as a scientific experiment of Balázs Gaál, an undergraduate student of information technology then, who meant to explore innovative methods for computer aided production of optimal dietary menus in 2002. All scientific work was done at that time, including research of the literature, elaboration of the multi-level organization of genetic algorithms, its application to the problem space, and the design of the fitness function with consultancy by dietary experts. The net result was a test program written in Microsoft Visual C++ demonstrating the feasibility of the idea and taking performance measurements. We call this software the zeroth generation **MenuGene** because later it became the takeoff point of further development. The work was presented on the 2003 annual National Academic Student League (OTDK) [7].

first generation

The zeroth generation MenuGene was incapable of anything beyond its designated scope (namely performance testing) and was not adjustable for other purposes. To remedy the situation, in January 2004 the author was asked to join the development, and shortly after the first generation was born. The objective was to create **GSLib**, a software library that would embody the optimization algorithm (above all the handling of multi-level genetic algorithms) on the top of an external generic GA library, **GALib**. We still have both of them, although the function of GSLib changed later; now that part is called **MGLib**. The original intention was to make provision for other problem spaces which could make use of MLGA optimization. Since then this has become a minor point, and today the software concentrates exclusively on dietary menu generation.

One of the urgent tasks was to write a networked program we ended up calling simply **daemon**, which would receive the desired parameters of a menu over network, then as a server would generate it using the new **GSLib** library. Before that a number of things awaited for being taken care of, therefore the first period of the project was spent on design decisions and factorization: class hierarchy, object relationships and recognized interfaces. Few code fragments of the consolidated library from that time has survived up to these days, but many of the concepts introduced then have remained.

By March 2004 **daemon** had acquired some functionality: it built the internal data structures essential for the menu generation from the database, but **GSLib** was not ready for that yet. It would have been vital for the project that the library become operational as soon as possible, but the early implementation had begun to show its fragility. It is common knowledge that software components can usually be divided into two broad categories: the ones (call them the *engine*) that are very specific to the task your program will eventually perform, and those (the *infrastructure*) whose reason to exist is to support the engine, not knowing anything about the details of the problem. Out of indolence the first generation **GSLib** employed a set of homegrown (and quite unsophisticated) classes for low-level data structure handling, whose maintenance got burdensome due to the increasing demands of the engine, the optimization algorithm. The infrastructure either had to be got right or be superseded by a third-party library.

The quest for a definite solution took several weeks and an experimental subproject (**migraine**). Finally a new external, template-based library, the **AAPL** got merged into the source code repository, and this marks the birth of the second generation. With **AAPL** one could define and manage collections of arbitrary types relatively easily, so our goal was achieved, it seemed. It felt so successful that in a week another external library, **TINYXML** landed in the repository in order to take over XML output production (formerly it had been done via unstructured **printf**(s)).

These merges gave significant boost to development progress. Early in June 2004 we could run the optimization algorithm of **GSLib** over random data via **randtree** (since then this program has become part of the **MGLib** test suite and it lives under the name **mlga**). Certainly we wanted to generate dietary menus, and it was **daemon** which connected the algorithm with the nutrition database. While that program had a developer-friendly command-line interface, the plan had always been to have **daemon** running silently in the background on a server machine, and let a client application display the results in a user-friendly manner. Thus **junkie** was born, a web application, which communicated with **daemon** over network. By August 2004 the system had become demonstratable, and some months later it was coupled with **CORDELIA**, the Department's lifestyle counselling system.

After these accomplishments development split into two branches. One of them aimed at reengineering **GSLib** to get rid of **AAPL**; it was necessary because few felt comfortable with the compromises induced by the unnatural linkage of that library. This effort (the **avl** branch), took a while to mature,

MenuGeneAdmin	so in the meantime the other branch engaged in tasks that did not depend directly on GSLib . One of them was the creation of MenuGeneAdmin for the dietary experts hired for professional consultancy, enabling them to remotely edit our nutrition database. Another subproject that emerged at that time was
ontologies	the exploitation of ontologies to represent the knowledge of a human dietary expert. Being done that would make it possible for humans to configure the optimization algorithm using terms and concepts more familiar to them than the solution-attribute model understood by the system.
database migration	Lastly, it turned out to be unavoidable to migrate and rework our nutrition database. Since day one MenuGene had relied on the database of CORDELIA , which was sold and abandoned in the course of events. Its data had been derived from sheets of a commercial vendor (QUADRO BYTE) and it had been managed by ORACLE on an unattended box. The problems were that doubts arose concerning the correctness of the information content, and that nobody wanted to dig into ORACLE and learn its advanced capabilities. It was time to work out an appropriate database schema, fill in with industry-quality content collected by the U.S. Department of Agriculture [20], and place it in the hands of POSTGRESQL , which the team knew better. These subprojectes, ontologies and migration, are still in flux, but definitely they are being worked on.
	Meanwhile the avl branch of the split was shaping, and evolved into a complete, future-proof rewrite of the existing GSLib . The reason behind the main surgery is that one needs to break certain psychological barriers to make large-scale incompatible changes. Once the taboos are down you might be better off getting through all of your long-standing intrusive ideas, because the next opportunity may come too late. In case of MenuGene among others the object model, the monitoring and error-reporting facilities, the XML writer underwent major improvements, many test programs were added and lots of documentation was written. For a detailed list of accomplishments the reader is referred to the TODO file. In particular, it was then when the engine was moved to its own library, MGLib , leaving all the infrastructural code in GSLib .
third generation	In mid-July 2006 the branch was ready to take over the mainline, and the third generation came into existence, making it feasible to implement some greatly anticipated features.
future	At the time of writing (May 21, 2007) most development power is spent on the database migration . The management is seeking for commercial investors, and the eventual goal is to make enough profit to get the project self-supporting. May we see whatever outcome, the author is confident that future developers will have a solid foundation that is worth building their innovations upon.

2.2 Qualitative goals

MenuGene was not developed with any particular methodologies in mind because none of the team members had been educated in these matters of software engineering. Nevertheless we cannot say the process was driven by individuals' inconsistent actions, either. In retrospect we can identify a set of

2.2. QUALITATIVE GOALS

Date	Subversion revision	Event
2004-01-19	r36	GSLib v1 (initial checkin)
2004-01-23	r50	large-scale OO reimplementatation
2004-02-23	r99	daemon checkin
2004-03-07	r157	daemon takeover
2004-03-11	r173	daemon2 born
2004-03-13	r183	daemon2 builds menu trees
2004-03-19	r199	major rewrites (starting to take over)
2004-05-16	r237	migraine
2004-05-22	r239	GSLib v2 (AAPL checkin)
2004-05-28	r289	TINYXML checkin
2004-06-03	r311	randtree is functional
2004-06-06	r314	mdb
2004-07-28	r456	daemon2 network mode
2004-08-06	r462	junkie2 born
2005-04-05	r511	CORDELIA integration
2006-04-07	avl	daemon3 , ttc
2006-04-24	avl	daemon3 : provision for CORBA
2006-07-13	r625	GSLib v3 transition complete
2006-09-06	r662	rules integrated
2006-10-04	r669	rulez born

Table 2.1: Milestones of project history

VASSÁNYI István	PostgreSQL, general advisor, academic contact
GAÁL Balázs	original idea, project executive, business contact, chief evangelist
VÉGSŐ Balázs	MenuGeneAdmin
MAYOR Zsolt	initial daemon code
CZIGÁNY Attila	CORBA interface for daemon
SZENTE Zsuzsanna	ontologies
HERCZINGER Viktor	PostgreSQL, system administration
the author	pretty much all and everything else

Table 2.2: Past and present project participants

extreme programming values that we took quite seriously, cutting out any others from the subconscious. This is similar to extreme programming (XP, see [3]), an agile software development methodology which is expressly suited for research projects having unstable requirements. It turns out many of the properties described below play an important role in extreme programming as well.

2.2.1 Maintainability

We postulate *no part of a system lasts forever*: one day or another it will be necessary to improve, fix, or investigate any given code fragment, no matter how still it used to be. To save the codebase from becoming an obstacle in its own development we strongly discouraged writing code on the assumption that once it is finished, it can be forgotten.

reasonable factorization In practice, one way to ensure maintainability was paying special attention to reasonable factorization. Whenever a new function was added, in the reviewing cycle we routinely examined: *Is it the appropriate place for that function? Wouldn't the implementation be simpler elsewhere? Or maybe is it worth splitting it up for better understanding?* If so, we tried hard to reach a reassuring conclusion.

elegant interfaces Related to code organization is the question of interfaces. As we all know, an entry point lies between the caller and the callee, their interaction depending on the encoding and decoding of arguments and return values, and the *mental* overhead of these processes can be significant (then painful). We realized this, and as a countermeasure we always sought to come up with elegant interfaces. Sometimes we found compromises unavoidable, because what callers appreciate is not necessarily comfortable for the callee. Other times we had to resort to code refactorization to be able to move, split, merge, generalize, or specialize interfaces, all because it appeared mental overhead would be less the other way. A set of well-pronounced guidelines on this topic can be found in [22] by the author of “glibc on diet”.

Gradual refactorization (both code- and interface-wise) is a key element of extreme programming. While it might be regarded as trivial day-to-day practice, we decided this task deserved more responsibility. If for no better reason, because it can break existing code considerably (which we did **several times**), but we have learnt being courageous (one of the principal XP-values) pays off.

consistent coding style XP also advocates good communication, which we (unknowingly) sustained by growing accustomed to a consistent coding style. *Developers express and interpret (communicate) program behavior by writing and reading code*, and they code the same thing very differently, in the terms of their personal vocabulary. We thought, at the cost of some soft restrictions we could spare a lot of mental overhead, because reviewers would not be forced to change their mindset when working on components written by someone else. Our coding style covered code formatting, markups, naming conventions, and the usage of language features. Once it had been documented, but later we saw it was better

to let people accommodate, rather than treating them as children who must obey their father's rules.

proper doc-
umentation

Finally, our professional experience had taught us not to neglect proper documentation. Following our **postulate of maintainability**, during the course of development we settled down on the principle *nothing in the codebase should be regarded as private asset* which need not be documented because it is shielded by publicly accessible layers. We attempted to make careful note of every detail which might not be obvious by just looking at the source, including expected entry conditions, side effects, relations with other functions (contributing to the access protocol of the component they are part of), and everything known to require further attention (TODO). Of course, there was the danger of watering down the source by excessive commentary¹. For us another utility of documentation was to gain feedback: *Does the behavior I have coded really make sense?* It occurred to us an idea is less likely to contain flaws if it can be phrased convincingly both in a spoken and in a programming language.

2.2.2 Correctness

automated testing

Besides the qualitative feedback picked up by writing documentation it was apparent we needed some quantitative measurements as well. Books on modern software engineering call for automated testing, an idea we adopted wholeheartedly for its promise of thoroughness and of saving human resources.

torture testing

One can automate his tests by several means. Knuth, for example, to verify the robustness of his **TEX** compiler crafted a special input file along with the expected output *manually* in the hope that the presence of bugs in the implementation would be visible either by a distraction in the output or by an unwelcome crash: [12]

The idea is to construct a test file that is about as different from a typical user application as could be imagined. Instead of testing things that people normally want to do, the file tests complicated things that people would never dare to think of, and it embeds these complexities in still more arcane constructions.

Knuth went as far as demanding no program shall claim itself a **TEX** implementation until it passes this test. He concluded: "This method of debugging, combined with the methodology of structured programming and informal proofs (otherwise known as careful desk checking), leads to greater reliability of production software than any other method I know." While we support this view, for **MenuGene** we needed a test method that relies less on the elaboration of test vectors.

unit testing

An appealing alternative was unit testing, so prevalent among Java developers that every book on Java programming, it seems, feels obliged to mention JUnit at least² (see [2] for example). Its application is quite diverse, but

¹Not a big threat as for today's standards.

²Giving the impression that Java does not know about any other testing methods—a statement we cannot accept faithfully.

we can make a couple of observations. First, unit testing (as a methodology) does not let testing become an afterthought, done (or forgotten altogether) after all user requirements have been met, because test cases and production code must be developed in parallel¹. Consequently, in the codebase there cannot be untested units. Exactly what constitutes a unit is largely unspecified [9], but the assumption of a one-to-one mapping between units and classes appears to be customary.

Most certainly we wanted testing to be an integral part of our software development lifecycle. On the other hand the tendency of unit testing *if each unit feels good, the whole program feels good* raised concerns about inter-unit relationships. By writing endless test cases one risks missing the forest for the trees, because operations of one object may *inadvertently* affect the state of other objects. Prime examples are global variables, shared resources in general and concurrency.

In the end what we have come up with may not be very different from some interpretation of unit testing. Our test suites consist of standalone programs, which are not built around any standard framework (although we managed to factor out some common code). Since our **build system** understands their invocation it is a matter of keystrokes to launch a test (or an entire test suite), and see if there is any regression. A test program tries to stress the implementation by use case basis, sometimes covering a subset of a class, other times more than one class. The test subject is put under stress by feeding it with *bulk, random*² input for a *long* time. The input is generated on the fly in such a way that frequently values are chosen near the limits of the input domain of the variables. IEEE refers to this method as *boundary-value analysis* [8, chap. 5]. For the input is ensured to be acceptable (though meaningless) the subject must not crash (see **section 2.2.3**).

Our line of thinking was the longer the subject is exposed the more likely we will hit a bug—if there is any. The advantage of the randomization is that we need not design test vectors to the minute details; it is enough to set the main directions, and leave the rest to chance. This technique is called *fuzzing* in security engineering, and its usage is naturally a compromise, but a good one at that in our assessment.

Stress testing (combined with measures to ensure robustness) performed well at demonstrating the behavior of the test subject is not incorrect, but its capability of gathering positive evidence is limited, because often the correctness of the results can only be checked indirectly. For this reason we advanced the method a little, and termed the result of our efforts co-testing for it is based on the simultaneous execution of two processes in a client-server manner, the client being the test program, and the server being the test subject. During testing, the client sends requests to the server, but *the operation is carried out by both parties*. At regular intervals the subject is asked to return detailed information about its state. As this information is internally

¹Even more, in a test-driven development pattern the test case for a unit is created *prior* to writing production code.

²Actually, pseudo-random. The seeds are always saved for the sake of reproducibility.

tracked by the test program too all the time, they can be compared to see if the subject has been malfunctioning. IEEE mentions this method under the name *back-to-back testing*.

It is important to emphasize the independence of the implementation of the test program. In our case not even the programming language is shared, yielding additional benefit of having insight into the problem the subject tries to solve from a different point of view. In an other project we experimented more with co-testing, and finally established a framework for trying the robustness and conformance of untrusted programs. [6]

2.2.3 Robustness

defensive programming One of our priorities was to write programs that behave well even under unexpected circumstances: if the input is intended (documented) to be acceptable (from the perspective of the callee) then, be it however extreme, it must be processed correctly; otherwise the error must be signalled to the caller very clearly. As we have seen in [section 2.2.2](#), much has been invested in the testing suite to probe this property of the implementation. Eric S. Raymond (ESR) characterizes robustness as “the child of transparency and simplicity” [16]. We have already touched upon these dimensions of software quality in [section 2.2.1](#); now we will review the measures we have taken to prevent bogus input *silently* producing bogus output¹.

appropriate data types ESR says “One very important tactic for being robust under odd inputs is to avoid having special cases in your code.” [16, *Ibid.*] We translated his advice into practice by making a good deal of special cases impossible. More closely, whenever feasible we declared variables with constrained types (like **unsigned** and **enum**²). Not only this gave a chance to the compiler to do static type analysis and catch misuse, but the point is to *confine the domain of the variables to the range that actually makes sense in that context*. A frequent use case is storing size or quantity, which obviously cannot be negative. Having not restricted their domain, they must be checked before usage—a special case we sought to fend off by ESR’s words.

consistency checkpoints Choosing the appropriate data type is a good starting point, but we knew it cannot work miracles, for example when the constraint is too complex to be formulated within the facilities of the type system of the target language³, or when the constraint is not known until run-time. As another layer of defense we strewed the codebase with internal consistency checkpoints everywhere we could think of. Similar in spirit to database constraints, these checkpoints try

¹This is not to be confused with GIGO (garbage in, garbage out), which is said when nonsensical (but acceptable) input about is fed to a well-meaning, correct program.

²The GNU C Compiler also has an `__attribute__((not_null))` extension, meaning a pointer declared such cannot be NULL. It is yet to be investigated, though propagation rules may wreck havoc on its usefulness. PostgreSQL 8.x has a more flexible mechanism to attach *custom* constraints to data types via `CREATE DOMAIN`. This feature fit very well in our database schemata.

³This is the case with Perl, which does not even distinguish between numbers and strings. In exchange it offers excellent validation capabilities.

to spot inconsistencies in the state of the program as early as possible, and once one is found, they bring on outright termination¹. For implementation we used both `assert(3)`-style short statements and separate sanity checking routines. After a while we came to using these safeguards as complementary formal documentation, and begun adding them purely for that purpose, even if they were known to be redundant. The price to be paid was the CPU-time spent on consistency checking, causing an order of magnitude slowdown. To cancel the effect, our **build system** can be **configured** to skip all the checkpoints, or just the most expensive ones of them.

input validation Robustness is a requirement of secure software. While MenuGene is not specifically security-sensitive (save for **junkie**, the web interface, which *is*), buying their agenda [23] is always a good idea, because one day any piece of code can end up in a context where security is a chief concern. One lesson people have (hopefully) learnt from web application development is the necessity of (program) input validation. [21] exemplifies a number scenarios when invalid input leads to incorrect behavior in subtle ways. It is important to realize that it does not make difference whether the input is invalid on (malicious) intention or just by mistake—the end result is the same unless this vector is taken care of properly.

error checking The textbook rule of thumb says you must also account for errors reported by subroutines (e.g. checking the return value). While it may sound too pedantic to bring up such a triviality in this document, visible evidence suggests developers still perceive error checking as a bothersome and unfruitful exercise². It is not that we were after illegible code for the sake of purity, but in our experience trading reliability for convenience is not a profitable deal in the long run.

graceful failure We have already alluded to various error handling strategies we employed in MenuGene: for example, halt on internal consistency error (bug) or on critical resource outage (i.e. insufficient stack space, or object allocation failure). The repertoire is broad; however, the point we made was to fail gracefully: do not crash the program, and try to clean up before exit. The latter clause may seem unrelated to the topic of this section, but an adhering component will appear to be more robust by not leaving mess behind even if it finds itself in trouble. Looks human psychology pervades software engineering.

2.2.4 Serviceability

Designing for failure may be the key to success is the title of an interview [13] conducted with Bruce Lindsay, IBM DB2 architect and RDBMS guru. Sections 2.2.2 and 2.2.3 showed how MenuGene is prepared to meet bugs and other failures. This section describes the built-in facilities that help developers and operators track down the source of failures.

¹The Linux kernel developers take the same approach, except that they seldom panic the kernel in consideration of fault tolerance. Instead they let the error signal propagate back to user space.

²Exceptions may help to hide this problem, but they will not take you much farther; see the next paragraph.

information gathering In such a situation it is probably information what the maintenance team appreciate the most: information about what the system had been doing before the error occurred, details about the context etc. Run-time information gathering is facilitated by the **gsmon** subsystem, which bears some resemblance with the **syslogd** daemon. Scattered in the codebase, just like **consistency checkpoints** there are calls to **gsmon** that inform about the progress of execution, current state, warn before entering and after leaving critical sections, or about anything the developer thought *might* be worth making record of some day. The logged information is filtered by the subsystem, and only the relevant pieces are presented. The source of the information (which component, function emitted it) is preserved as well as the call stack at the time the message was logged (with a little aid).

error stack Components often fail due to the failure of another component. This case the error propagates back in the dynamic scope of the program. The error could be logged by every component involved in the chain of events. While **gsmon** is perfectly suitable for this usage, we chose to add an error stack manager, **gserr** to assist handling the chain of errors¹. In a typical use case when an error is encountered it is added to the queue, and the control is returned to the caller (indicating the failure). Depending on the error handling strategy of the caller, it may give up too, retry the operation, or just acknowledge the error and proceed. Not all of these cases needs the error to be reported. **gserr** can clear the queue either reporting or just forgetting its contents, depending on the judgment of the caller. This way the full chain of events can be retrieved if necessary, but they will not flood the logs if they are harmless.

service interfaces Both logging and the error queue are active in normal operation mode. **MenuGene** also has service interfaces, which come into play in actual debug sessions. Some of the interfaces function as hooks. These are placeholder NOP instructions which you can set a breakpoint on, allowing, for example, to trap exceptions. (This is not possible otherwise in C++.) Other service interfaces exist to instruct various components of the system to produce extra debugging information (like statistics, for profiling).

2.2.5 Care

In a way care can be regarded as a meta-value: having high standards of the values discussed so far infers care about the quality of the software overall. Extreme programming has added respect as the last of its own values (the original proposition did not have it). XP practitioners respect their work by pursuing high quality and looking for the best design for the solution at hand.

prototyping Unfortunately quality does not come for free. Often we found ourselves spending days or even weeks on interface design, refactorization, and prototyping, coding several possible variations to choose the best one. Done with that in some cases we managed to integrate prototypes into our **co-testing framework**, saving materia that otherwise would have been consigned to oblivion.

¹In Java, we talk about exception chaining.

2.3 Development environment

2.3.1 Platform

Linux Before the author joined MenuGene had been developed and run on Microsoft Windows. Shortly after the team shifted to Linux, and has remained with it ever since. The reason behind the move was simply that we had more experience with Linux, and we wanted to support that platform anyway. Currently the system neither compiles nor runs under Windows, but porting should be fairly trivial, because care was taken not to overuse platform-specific features. Beside the two mainstream desktop operating systems, no other has ever been considered.

2.3.2 Programming languages

The bulk of MenuGene is written in C++, mixed with some embedded SQL, surrounded by Perl and PHP programs, and glued by shell scripts.

C++ The choice of C++ was predetermined by the external low-level genetic algorithm library, GALib, which was also written in C++, and replacing that library was out of question¹. Since the engine of MenuGene and GALib need to cooperate closely, choosing another language would have introduced unacceptable risks of instability. C++ is a usual compromise between execution speed, development speed, and portability. For MenuGene the first one is the deciding factor, for genetic algorithms are truly CPU-intensive.

migraine As noted above, C++ is usually a compromise. Many feel dissatisfied with its misfeatures and unpredictable behavior (see [10] for a well-founded case against C++). [5, 19] list *hundreds* of obscure situations when the language acts far off the expectations—not one of them we had the pleasure to deal with. In an attempt to back out of the jumble of C++ we created a short-lived branch, **migraine**, to probe into Objective C. The subproject was canned soon, but the gathered experience bore heavy influence on the memory management subsystem of the second generation **GSLib**.

Perl The need for a higher level language was stressing. As an alleviating resolution we incorporated Perl to write auxiliary and helper applications, such as GUI visualizers, like **rulez**. Perl is very good at handling data structures, functional programming and object oriented programming². Its characteristics makes it ideal not only for prototyping, and the author expects to see an increasing portion of MenuGene being written in Perl, if the development of the system is to be continued once.

¹The project management pled to its alleged widespread scientific usage.

²Despite the near-complete lack of special syntax for OOP.

2.3.3 Compilers

Contrary to popular misconceptions the C++ compiler of the GNU Compiler Collection (`gcc`¹) is not the only one available on Linux. A serious alternative is the Intel C/C++ compiler (`ICC`), which is said to generate faster, superior code. However, we decided to stick with `gcc`, which we understood very well, and reserved `ICC` as an option for later evaluation. `gcc` has a wide variety of vendor-specific extensions which we sought to exploit aggressively (grep the source for “gcc-specific” to see the whereabouts). As neither compiler-neutrality nor purity was an objective explicit dependency upon these extensions only made life easier for us. Arguably, it is a bad idea to get into a vendor lockin deliberately, but in this special case there is a way out: the `ICC` is rumored to be compatible with `gcc` to such an extent that even the Linux kernel has no problem with it. [24]

2.3.4 Build system

By *build system* we refer to the framework that utilizes your compiler toolchain to automatically compile source files and produce an executables. (See [8, p. 114] for a better, more abstract definition.) Self-contained IDE:s (like Microsoft Visual Studio and Eclipse) come with their own build systems, storing related settings in *project files*. Unlike with the `gcc`, we definitely did not want to depend on a particular IDE to work on the system—people are different, after all, and an editor that fits one’s needs may be a pain to use for others.

`MAKE` Instead the build is controlled by GNU `MAKE`². Making use of the macroing capabilities of the GNU version (`\$(call ...)`), all the building logic is sourced out to one specific location (the `BUILD` directory). Developers can usually get on by copying an existing `Makefile` and adjusting a few variables as documented in `Makefile.vars`.

2.3.5 Software configuration managers

A software configuration manager (SCM) is a program that, put very simply, tracks the history of the codebase and allows team members to share their improvements with each other. When a change is done the developer must notify (*commit*) the SCM. [8, chap. 7] has a comprehensive review of all related concepts.

IEEE attributes critical role to SCM in the software maintenance process. [14] It is no coincidence that the codebase of MenuGene has been under version control since the very beginning. Initially `CVS`, the de facto tool of not-

¹By the acronym we will refer both to the GNU C and GNU C++ compilers. See [18, p. 3] for other meanings of the acronym.

²It must be noted that there exists many other build systems, such as `CONS` and the infamous `JAM`. It is only that the extended capabilities of GNU `MAKE` made it a satisfactory solution for us.

for-profit development efforts was responsible for SCM. It was neither particularly featureful nor reliable; especially its inability to follow file and directory renames was disappointing.

SUBVERSION When it became apparent that [CVS](#) loses commits we decided it was time for departure. A potential successor was [MONOTONE](#), a distributed SCM supporting disconnected (offline) operation. This mode was quite welcome by the author, who did not have Internet connectivity at home at that time. We considered [MONOTONE](#) briefly, but the team rejected it on the grounds of being too different from what we were used to. Eventually [SUBVERSION](#) took over [CVS](#) at the same time when the [third generation](#) of [GSLib](#) was merged. The transition period lasted no longer than a week, and at the end, much to our reassurance we managed to preserve the history of the codebase.

2.3.6 Developer tools

Out of respect to developers' freedom we did not discipline what CASE tools are to be used for everyday duties, except when agreement was absolutely necessary (e.g. the choice of SCM). As a matter of long-established preference the author keeps on using [VIM](#) for editing, [GDB](#) for debugging and [CTAGS](#) for generating source code cross-references. Other team members settled on [ECLIPSE](#) and are content with it.

homegrown utilities An often neglected segment of software construction is the creation of homegrown utilities that automates repeated tasks (for example, disassembly of a dump file) or makes the developer's life easier otherwise. We did not make trouble of it and wrote new tools whenever necessity came in the same vein as any other code. The [next section](#) provides a detailed listing and description of these tools.

2.3.7 Database managers

ORACLE Part of the MenuGene system is an [ORACLE](#) database manager, which provides efficient access to nutrition data for other system components. (See [figure 1.3](#) in [chapter 1](#).) The database schema and contents dates back to days when the author was not affiliated with the project yet, when MenuGene was planned to be a subsystem of [CORDELIA](#), the Department's lifestyle counselling system. (A weak linkage has been [fixed](#), but only for demonstration purposes.)

criticism It is obvious now, [ORACLE](#) is an overkill. Its capabilities are greatly underutilized, and the database schema has stagnated beyond all bearings. MenuGene has proven not to be a data-intensive application anyway, as its data supply scarcely exceeds a few megabytes.

POSTGRESQL In response to the first and second lines of criticism, the project is drifting away from the world's leading vendor of information management software towards [POSTGRESQL](#). One of its key advantages that led to the decision is its remarkably comprehensive and readable documentation¹. The transition

¹Though compared to the APIs and manuals of [ORACLE](#) anything may seem 'excellent'.

is also a good opportunity to rethink and (at last) document the database schema—a long-due debt to repay.

2.3.8 Documentation

source code
commentary

MenuGene does not have any *formal* documentation, largely because there was no up front design in the first place; this document is written in the hope to **fill the gap**. This is not to say there is not an explanatory word in the codebase, that would be against our **qualitative goals**. Rather, most of the information is in the source code commentary. There was an early attempt to make it compatible with **DOXYGEN**, which would then produce nice HTML sheets just like **JAVADOC**, but the effort has lost interest. In the **doc** subdirectories there are some complementary files for the record of notes not fitting elsewhere. The draft **POSTGRES** database schema is kept in **src/daemon/doc/postgre.sql**. The description of various database objects (**TABLEs**, **DOMAINs**, **VIEWs** etc.) are also available on-site in **COMMENT** blocks, so SQL development environments can access them.

TODO files

Bug tracking (*defect tracking*) is also handled informally. Once we had a **BUGZILLA** to put bugs on public exposure, but it turned out not to help productivity. What we needed instead was a registry of missing and broken features ordered by relative importance to know what areas call for attention. This list is maintained in the **TODO** file. By recalling past version of this file via the SCM, one can estimate the accomplishments and limitations of the system at any given time.

2.3.9 Quality assurance tools

memory-
related bugs

Several tools have been put in service to assure the **correctness** and **robustness** of MenuGene. While they differ in specific uses, all of them are employed to unveil memory-related bugs. A common detrimental side-effect of these tools is the significant slowdown they cause, making their application difficult during **stress testing**.

- **ELECTRIC FENCE** is a library that transparently replaces all routines of the **malloc(3)** family to catch *buffer overruns* as soon as they happen. The technique is solid, but wasteful: even a single byte of allocated memory will occupy kilobytes. (The formula is not linear.) Worse, on Linux the size of user address space will drop down to a fraction of the normal limits. This means the program will not be able to claim more than half gigabytes of memory, no matter how much is available.

Nevertheless, the help of **ELECTRIC FENCE** has been invaluable. Years ago we forked a version under the name **ElectricTense** for private enhancements. Since then it has grown completely independent, and today it is part of **QBOT** [6].

- **CCMALLOC** is another transparent library, which we primarily used to detect *memory leaks*. The library can tell where the unreleased piece of memory was allocated in the source, hinting at the faulty data path.
- **VALGRIND** is a virtual machine which observes program behavior on the machine instruction level, and draws inference from that. We used it to spot *references to uninitialized variables*.
- **mdb**: One of our homegrown utilities to *profile memory consumption*.

Java developers will notice their choice of language offers these services out of the box. While there is no getting around it, it must also be noted the tools listed above are not the king achievements of the 21st century realm of C. For one, the copyright of **ELECTRIC FENCE** dates back as far as 1987—eight years before the burst of Java.

2.4 The codebase

2.4.1 A walk through the repository

In this section we will review the contents of the MenuGene repository. We will concentrate on role and functionality, allowing for few technical details. These can be followed up at the referred pages, or in the source code itself if the component is not discussed further in this document.

- **BUILD**

This directory contains the common files needed to **build** the system.

- **eclipse**

This script is invoked by **make eclipse** to get the source browsing capabilities of the **ECLIPSE** IDE fully functional. It does so by linking header files to another place in the file system. **make xclean** undoes the effect, making the repository pristine again.

- **Makefile.vars**

This file is sourced by every **Makefiles**. It describes the build environment, such as the desired compiler to use. These parameters are intended to be customizable to fit local needs.

- **Makefile.macros**

Sourced internally by **Makefile.vars**, and describes the actual operating system commands to execute in order to build a part of the system. It should be general enough to be applicable unmodified to any Unix-like systems.

- **Makefile.rules**

This file is also sourced by every **Makefile**. It consolidates all the information from the **Makefiles** described above, and eventually tells **MAKE** when, how and what to build.

- **GALib**

Complete source code of **GALib**, the external low-level genetic algorithm library **MenuGene** relies on. It is included in the repository to be at hand if the system happens to need a feature not provided by the upstream version. It has been brought up-to-date with newer compiler standards, and the source layout has been rearranged to be more consistent with the rest of the repository.

- **GSLib**

This directory contains the source code and test suite of **GSLib**, the infrastructural library of **MenuGene**. In more precise terms, in this library reside the code of the **object model**, the generic complex **data structures**, the **XML output driver**, the **logging**, and the **error reporting** subsystems.

- **MGLib**

In this directory are the source code, documentation and the test suite of the **MGLib** library, the engine of **MenuGene**, responsible for the generation of optimal menu plans; this is where the optimization algorithm lives. This library is not an abstraction layer above **GSLib**, because client applications need to interfere with both of them.

- **migraine**

An early attempt to **reimplement** the optimization method in Objective C. By now it has lost all of its functions, but we believe the idea is worth keeping in mind.

- **perl**

This is the location of all sort of Perl modules **used** by Perl programs all over the repository.

- **Adios**

Adios is the author's bundle of custom Perl modules for random utility purposes. It is not maintained as part of **MenuGene**; what is in the repository is merely a copy of the (unreleased) upstream.

- **CoTest.pm**

Provides common functions for **co-testing** parties. It helps in establishing a communication channel between the tester and the subject, and in comparing the results of the parallel execution.

- **GSDB.pm**

This is an abstraction layer which makes accessing the **MenuGene** database conceptually more natural for the programmer.

- **Rules.pm**

Prototype implementation of the **rule matching** function of **MGLib**. In part, it is used for testing.

- **MyPrima.pm**

This is a collection of widgets and subsystems that extends the functionality of the **PRIMA** GUI toolkit, which is required by some of the **MenuGene** programs. Among the additions are a drag-and-drop manager, an undo manager, a flexible scrollbar manager, and a new layout manager.

- **daemon**

There is the source code of **daemon**, a client program that takes user criteria, loads some data from the database and generates appropriate menu plans with the help of **MGLib**. It has a network interface, making it suitable to be used as a mid-level tier behind a user-friendly presentation layer.

- **postgre.sql**

This is the draft of the new **POSTGRES** database schemata of **MenuGene**. The database that the system has today unfortunately lacks documentation. Until the replacing database manager takes place this draft can be consulted to get an impression about the conceptual design of the current schema.

The following files appear in many other parts of the codebase, so it is worth casting a few words about them.

- **.depend**

This file is generated by **make dep** and used by the **build system** of **MenuGene** to track dependencies between source and object files.

- **.ccmalloc**

This is the configuration file of **CCMALLOC**, one of our **quality assurance** tools.

- **.suppress**

Configuration file of **VALGRIND**. Both **.suppress** and **.ccmalloc** are used to instruct the associated tools to try to keep out false positives from their failure report as much as possible.

- **.gdbinit**

Initialization file of **GDB**, the author's choice of debugger. The commands in **.gdbinit** make the debug session more convenient and more efficient.

- **junkie**

Source code of **junkie**, the chief end-user web interface of **MenuGene**. It relays user input to **daemon** through network and presents the results in a neat form.

- **rulez**

rulez is a GUI program for assembling **rules** for MGLib, guiding it in the selection of menu plans based on user taste and other preferences. Originally, it was intended to be just developer tool to create test vectors for the associated functionality, but we believe its user interface has been polished up to the level of meeting not-so-high end user expectations, like drag'n'drop and unlimited depth undo. **Figure 5.5** demonstrates the program in operation.

- **BITS**

This is the developer's private quarters. The code in this directory – our **homegrown tools** and other asset – is required neither for building nor running the system, but can be of value when developing it. Demonstration screenshots can be seen at the **appendix**.

- **asciitree.pl**

Text-mode XML visualizer. It reads XML files and draws flat ASCII diagrams showing the hierarchy of XML elements. This tool is used to inspect the XML output of **daemon**.

- **coxi.pl**

This is a GUI XML browser, useful at exploring larger XML structures than **asciitree** is suited to. We had evaluated a number of mature XML browsers before writing **coxi**, but all of them felt so heavyweight that we decided they were not worth the disk space for our modest needs.

- **baggy.pl**

Along the lines of **coxi**, this is the browser of the MenuGene database. It displays the **relationships** between solutions, attributes and solution sets. **baggy** was our first GUI tool, and in part it served as a pilot project probing into the GUI potentials of Perl—with salient success.

- **dolly.pl**

Downloads the complete MenuGene database and saves it in a compact binary file which **baggy** and **rulez** can read, so they can operate disconnected from the database manager.

- **subborn.pl**

subborn is a **SUBVERSION** repository browser, with special focus on convenient history traversal. In a mouse click it lists what the currently selected directory contained at the previous revision, and with the help of the **VIM** editor it can depict the differences between two revision of the same file in unmatched quality. To our astonishment, we found no standalone applications among the mainstream **SUBVERSION** GUI clients that had similar features.

- **PROTO**

Prototypes for various parts of the system used to live here. As time passed and they had fulfilled their purpose, most of them have been removed from the repository. The only one remaining is **cow.pl**, modeling the implementation of the optimization algorithm of **MGLib**.

- **demos**

The diagrams in this directory have been generated by **cow.pl**, and have been used to examine and demonstrate workings of the optimization algorithm as implemented in the prototype. It has been planned to add a real-time monitor to **daemon** with similar diagrams on display. This work is yet to be done.

- **TOYS**

Developer playground. The code fragments here are not meant to do anything useful, they are just samples of how to accomplish a particular task.

- **mdb**

mdb is a small library for memory profiling. In run time it logs all memory operations (**malloc()**, **realloc()** and **free()**) to a binary file which can be analysed with **mdbstat.pl** later. In this data one may discover patterns in the memory usage of her program, and associate them with operations being executed at that time, helping code optimization.

- **ttc**

Total tracer is a C++ add-on tracing function and method calls during a program's lifetime. Similar to **mdb** these events are recorded in a binary log file, which is decoded by **ttc.pl**. The output is listing of what functions in what order from where were called during the tracing. Naturally, this list can grow quite long, but the decoder takes great pains making it as compact and readable as possible.

2.4.2 Dependencies

MenuGene makes extensive use of external tools and libraries. The following packages are sufficient to build and run all of the components of the system.

Package	Remarks
GNU MAKE a minimal POSIXy environment the GNU C++ Compiler	any recent version will do, but it must be the GNU variant required by the build system v3.4 is what we are using; v3.0 will not do, v3.3 might do, v4.0 is known to generate incorrect code
PERL the Oracle Client Interface the PRIMA Perl GUI toolkit	either v5.6 or v5.8 will do but the former lacks a few mod- ules, which might need to be installed separately then either v9.2 (9i series) or v10.2 (10g series) will do; for the least trouble we recommend our stripped OCI distribution required by rulez and subborn ; since Linux distributors seem not to offer binary packages, we have rolled our own , which is as easy to install as unpacking a tar archive

Also, one may want to obtain the packages below in case she intends to carry on the development of MenuGene and put our [in-house tools](#) in working order.

Package	Description	Needed by
GOBJC ZSH	the GNU Objective C Compiler the Z shell (Unix command line inter- preter)	migraine mdb
DBD::ORACLE	ORACLE backend for DBI, the Perl Database Interface	GSDB
GTK	Perl bindings to the GTK+ GUI toolkit library	baggy
MATH::RANDOM	Perl module generating random devi- ates; used to generate test vectors	co-testers, cow.pl , cross.pl
MATH::COMBINATORICS	Perl module enumerating combinations and permutations; used to generate test vectors	cross.pl
HTML::PARSER	One-pass HTML processor module for Perl; used to read XML files	asciitree
HTML::TREE	HTML parser module for Perl; used to read XML files	coxi
DATE::PARSE	Perl module interpreting textual repre- sentations of dates	subborn
GRAPHVIZ	Graph rendering software package	relations.pl , cow.pl , cross.pl

2.4.3 How to build it

Building the executables of the system is relatively straightforward. The first step is to adjust the settings in [src/daemon/daemon.pc](#) under the heading

daemon “Ora connectinfo”. Based on these settings knows **daemon** where to find and how to log in the database manager. We have not made these values run-time configurable because our server has never changed. Nevertheless it would be a reasonable improvement.

It is wise to verify the `$ORACLE_HOME` environment variable is set correctly.

OCI This variable belongs to the **OCI** libraries, and is supposed to point to a directory where C header files and shared libraries can be found underneath. If you have our **OCI distribution** installed, sourcing `/usr/local/bin/oracle_env.sh` will set the variable correctly.

configuration There are some other configuration options and tunable parameters in `src/GSLib/include/gslib/config.h`, but we figure that their present values are just right. It might be desirable to change `CONFIG_DEBUG` of `src/BUILD/Makefile.vars`, which controls whether to build debugable or production object code. See the files themselves for the complete list of available settings.

MAKE Now **MAKE** is ready to build the executable parts of MenuGene. Invoked from the root of the repository (`src`) it will take care of everything automatically. Components can also be built individually by issuing the command from their directory. The code should compile without warnings when `CONFIG_DEBUG=full`. Below that debug level (`half` or `none`) the ever-cautious **GCC** will take notices about possible instances of uninitialized variables; these warnings have all been verified to be false positives and can safely be ignored. The following operations (*targets*) are available:

- **make** or **make all**:

Just build everything in the current directory.

- **make dep** or **make .depend**:

Recalculate object file dependencies. **MAKE** will into all source files (*sources*) to see what other files (*includes*) do they refer to. This is necessary for the build system to determine whether an object file (*object*) is out of date and needs recompilation. You should **make dep** whenever a new include is added to any of the dependencies.

- **make <source>.o**:

Build the named object if it does not exist or any of its dependencies has modified since the last build.

- **make <program>**:

Likewise.

- **make tags**:

Index all source files with **CTAGS** for quick crossreference.

- **make eclipse**:

Clone header files where **ECLIPSE** finds them. This is necessary because of a **GCC** extension (`-remap`) that **ECLIPSE** does not understand.

- **make clean:**

Delete all compiled files from the current directory and downwards.

- **make xclean:**

Delete all reproducible files. Implies **make clean**.

The test suite of **GSLib** and **MGLib** are not administered from the top-level directory; see their sections to learn how to run them.

deployment The eventual target of **MAKE** is the executable of **daemon**. Its deployment is very simple: copying that single file to any box where **OCI** has been installed will suffice. Deploying the other components of the **MenuGene** can be just as easy as well, but so much depends on the actual operating environment that the matter is best left to experienced system administrators.

Chapter 3

Implementation details

GSLib is the infrastructural library that both **MGLib** and **daemon** relies on. It is a pool of functionalities not directly related to the problem MenuGene tries to solve—optimal dietary menu generation. Its name (“Genetic Solver Library”) originates from the **time** when we thought bundling everything into a single library would be maintainable. Since then the infrastructural needs have grown, and the engine has been granted its own library, MGLib.

GSLib is written in C++, and (though not for the faint of heart) makes heavy use of templates. Generic programming (templates) is a complex addition to modern C++, with startling syntax and peculiar semantics. One of its toughest aspect is the point of template instantiation. Basically, the question is where in the source code instantiate templates in order to reduce compilation time and object code size. **GCC** offers several options [18, Template Instantiation]. Since neither of the factors mentioned was our concern, we decided to follow the Borland model and “Do nothing [extraordinary]. Pretend g++ does implement automatic instantiation management.” In a nutshell, in the Borland model template definitions must be visible to all sources that instantiate them, and the compiler emits object code for every instance for each file being compiled. In our case the definitions are in the files of **src/GSLib/inline**, which are **#included** whenever necessary. The drawback of this approach is that header files become interdependant upon each other—a phenomenon well-known to C programmers. To cut the long story short, we have come up with an organisation in which header files are split into multiple files – declarations, definitions, macros, prototypes – and every one of them only **#includes** those parts that are really needed, ordered such that the dependency cycles break open. This mechanism is transparent to **.cc** source files. The commentary in **src/GSLib/include/gslib.h** presents an illustration of the problem and the solution.

Originally, templates were introduced into C++ to save developers from the ‘evil’ C preprocessor, and increase the type safety of the language. It is ironic that the C++ standard committee seems to have fallen between two stools in their effort. It is our persuasion that the preprocessor has a place in the developer’s toolbox, and we did not refrain from using it, despite its deteriorated reputation. As a result, GSLib has dozens of macros, which are



template in-
stantiation

header or-
ganisation

macros

essential for the workings of the library. A quick survey of `gslib/macros.h` will reveal that very few of them could possibly be replaced by templates. In fact, many of the macros are employed to circumvent the all too disciplining nature of C++. Definitely, this is a case when good intentions fire back on consumers.

`macros.h` `macros.h` is our store of general macros not belonging to any specific subsystem of the library. They define shorthands for common expressions, help in defining classes, simplify casting etc. We cannot review here all the 69 of them, but the documentation in the file should give adequate coverage.

3.1 Object model

`GObject` `GObject` is the core object model of `GSLib`. The model defines a set of conventions that every `GObject` must conform to, and another set of methods by which any conforming object can be manipulated. The semantics and the implementation may be familiar from other frameworks; in particular, many ideas were borrowed from Java and Objective C.

3.1.1 Basic facilities

`CObject` `CObject`¹ is the ancestor of all `GObjects`. The common operations are propagated via inheritance (whenever possible), and the conventions are enforced via abstract methods (whenever possible, see `usage`).

reference counting First and foremost what a `GObject` ‘knows’ is the semi-automatic management of its own lifecycle. It is well-established that C++ does not reclaim dynamic memory automatically. The problem is that it is left upon the shoulders of developers to find out when it is safe to `delete` an object. This is not trivial, because many other objects may hold reference to the named one. `GObject` helps to alleviate the problem by providing little-overhead means of explicit reference counting. The principle is simple: a counter in every `GObject` maintains the number of referrers to that object. Referrers express their (loss of) interest in a `GObject` by manipulating that counter.

- `retain()`:

Tells the receiver `GObject` that the caller got hold of a reference to it. We speak about *joint ownership* if an object has more than one referrers.

- `release()`:

Informs the receiver that one of its referrers has ceased caring about it. When its last referrer has gone, the object is destroyed.

- `is_protected`:

Setting this field `true` indicates that the object has a permanent referrer, which will stay there no matter how many times `release()` is called.

¹All identifiers defined in the library are in the `gslib` namespace. For the sake of readability we omit the prefix in the text.

Consequently, the reference counter will not be allowed to reach zero, and the protected object will not be destroyed then.

A newly created `GObject` is assumed to be owned by its creator. When an object is destroyed, it must explicitly `release()` every dynamically allocated `GObjects` that it has reference to. Inline (non-dynamically allocated) `GObjects` must not be `release()`d by their container object, and at the time of destruction they must not be referred by other objects.

may change During its lifetime ownership of a `GObject` ownership transfer, for example when it is passed to a setter method. Technically, exactly nothing happens then, but semantically it implies that one reference of the caller to the object will belong to the called method from now on, as if the callee `retain()`d and the caller `release()`d the passed object. It is not specified in what circumstances the ownership of a passed object is transferred, it depends entirely on the called method. For setter methods the usual convention is to take ownership, while getters keep their ownership of the returned object.

protection Sometimes the forceful takeover may be unacceptable. For example, if the caller would continue to use the passed object, but somehow it ends up destroyed by the callee. This is when `is_protected` can help.

duplication Besides reference counting `GObject` has other things to offer with respect to object duplication. An object can be in *has* relation with other objects, which in turn may *have* other objects etc. When making copy of an object it is desirable to specify how deeply the hierarchy should be duplicated, and from which level should it be a clone of the original. The distinction between clones and duplicates is clear: if A is a clone of B then changes to A are visible in B, while if A is a duplicate then B is unaffected (though initially they have the same contents). [Figure 3.1](#) will clarify the meaning of *depth*.

- `ctor(mate, depth):`

Every conforming `GObject` is required to have a copy constructor with a prototype like this. The `depth` parameter specifies how many levels of the contents of `mate` should be duplicated; above that level contents are cloned.

- `dup(depth):`

Returns with an object of the same type as the receiver. [\[5\]](#) calls it a *virtual constructor*. The new object is initialized with the receiver. `depth` has the same meaning as at the copy constructor.

- `copy(depth):`

The same as `dup(depth $-$ 1)` for positive values of `depth`, otherwise the receiver is returned.

- `is_shared():`

Returns whether the receiver has multiple referrers.

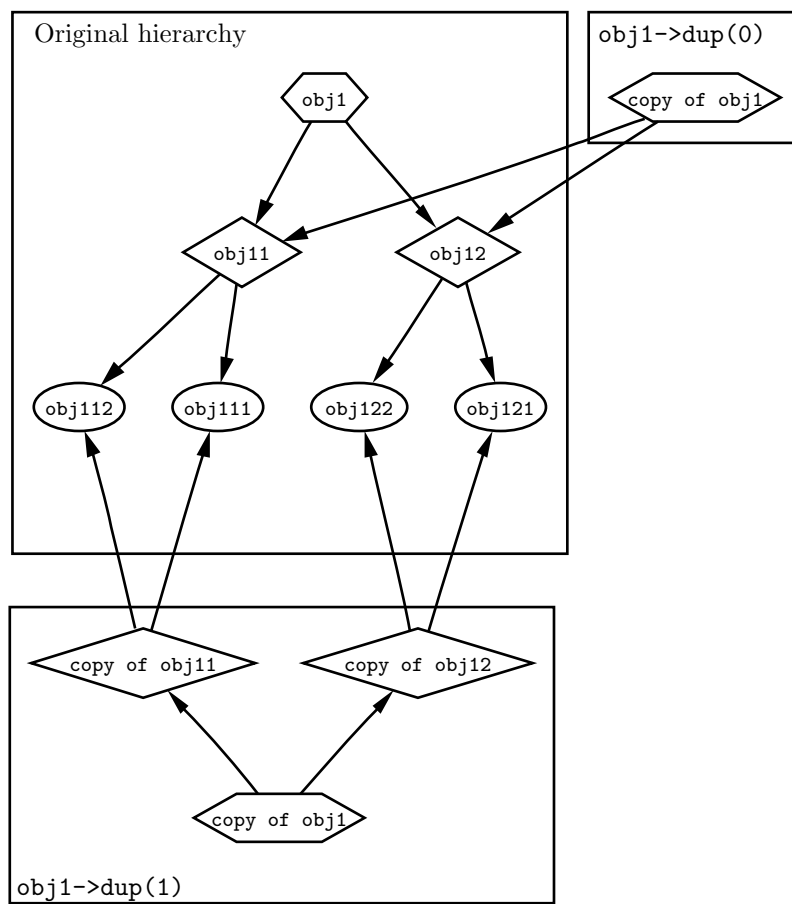


Figure 3.1: The effect of `dup()` with depths 0 and 1 on a three-level object hierarchy.

- `unshare(depth)`:

`dup(depth)` the receiver if it `is_shared()`.

`dup()`, `copy()`, and `unshare()` return pointers of type `CObject *`. All three of them have a capitalized pair (`Dup()`, `Copy()`, `Unshare()`), which behave exactly like the lowercase versions, except that they return upcasted pointers. In addition to these operators `GSOBJ`s can be moved around with the macros `GSOBJ_{COPY,SET,ZERO,CLEAR}()`. They are described in [GSOBJ/macros.h](#).

Finally, a conforming `GSOBJ` *may* choose to respond to a series of other interfaces requests. Implementing them is not compulsory—`CObject` only defines them to make it syntactically valid to invoke the methods in question of *any* `GSOBJ`, regardless whether it actually implements them. This can be considered a workaround to a fundamental flaw in the ‘object orientation’ of C++.

- `compare(mate, flags)`: (interface `IComparable`)

Asks the receiver to compare itself to another object from the same class, honoring the semantics laid out in the [compare namespace](#).

- `serialize(options, base)`: (interface `ISerialisable`)

Asks the receiver to give a [GSXml](#) representation of itself. `options` is a collection of arbitrary options whose meaning is up to the receiver. `base`, if defined, shall be the parent of the XML node of the XML representation of the receiver class.

- `print(stream)`: (interface `IPrintable`)

Asks the receiver to give a byte representation of itself, and print it to `stream`. Details of this representation are unspecified. At the moment `GSlib` uses XML exclusively, but you may choose any encoding you wish.

There are other interfaces that a `GSOBJ` may implement, which are not part of the core `CObject` class. See [GSOBJ/definitions.h](#) for further possibilities.

usage At the end let us examine how to define a class conforming to the `GSOBJ`-object model. As mentioned earlier in the section many constraints cannot be expressed with the inheritance mechanism of C++. For this reason we need to use macros *within* the class definition. Unfortunately, this practice tends to confuse the source indexing feature of advanced IDE:s. The following listing illustrates the procedure. For background, please see the description of the macros involved.

```
class SomeClass :
    /* GSOBJs must be a descendant of CObject.
       */
    public CObject
{
public:
```

```

    /* Required by GSOBJ_DEFINE_METHODS(). */
    GSLIB_DEFINE_TYPE_THIS(SomeClass);

public:
    /* Declares the copy constructor as required
       by the
       * GSObject object model. */
    GSLIB_DECLARE_CTOR_COPY(SomeClass)

public:
    /* Defines common GSObject methods which
       cannot
       * be propagated via inheritance. */
    GSOBJ_DEFINE_METHODS();

public:
    /* The following items are optional. */
    /* Declares compare(), if you wish to provide
       * a real implementation. */
    GSOBJ_DECLARE_COMPARE();

    /* Declares serialize(). */
    GSOBJ_DECLARE_SERIALIZE();

    /* Declares print(). */
    GSOBJ_DECLARE_PRINT();
};

```

3.1.2 Copy on write

There are situations when having multiple *private* copies of the same object is desired. This could be achieved easily by creating and initializing the object, and **dup()**ing it as many times as required. Then changes to one duplicate of the object would not affect the others, but the cost in terms of system memory may be intolerable. We faced this very problem when implementing our **genetic model** in **MGLib**, and our solution is based on the observation that often the requirement of having a private copy can be downgraded to “give me a shared reference to the object which stays as it is, and give some means by which I can make it entirely my own as necessary comes”. In less informal terms this technique is called *copy on write*.

protocol The protocol is as follows: every interested party gets the same reference to the object (let us call it a *cow*). The parties are registered as *watchers*. They may query the object as long as it remains unchanged. When a watcher needs to change the object the other watchers are notified beforehand, so they can **dup()** it. At the end there will be two copies of the object: the original one in the private possession of the watcher which wanted to change it, and a

duplicate, being shared among the remaining watchers. Of course the duplicate will have one fewer watchers than the original object had. [Figure 3.2](#) depicts the course of events.

For the notification mechanism to work watchers must implement the `INotifyable` interface (declared with `GSOBJ_DECLARE_NOTIFY()`). This interface is supposed to `dup()` the cow if necessary, and replace the watcher's reference to the original object by the new one.

cows A cow object must be a descendant of the class `CCow`, which contains the registry of watchers and the code for the propagation of change notifications.

- `add_watcher(referrer):`
Adds `referrer` to the list of watchers, which will be informed when the object is about to change.
- `del_watcher(referrer):`
Removes `referrer` from the watcher list of the cow. `referrer` remains a referrer—no `release()` is implied.
- `notify_watchers(originator):`
Issued by a watcher before it alters the state of the cow. By the time the method returns, the cow is guaranteed to have become private to the caller.
- `cow_depth:`
Specifies how deep the duplication of the cow should be on write. The default is to make the deepest copy possible.

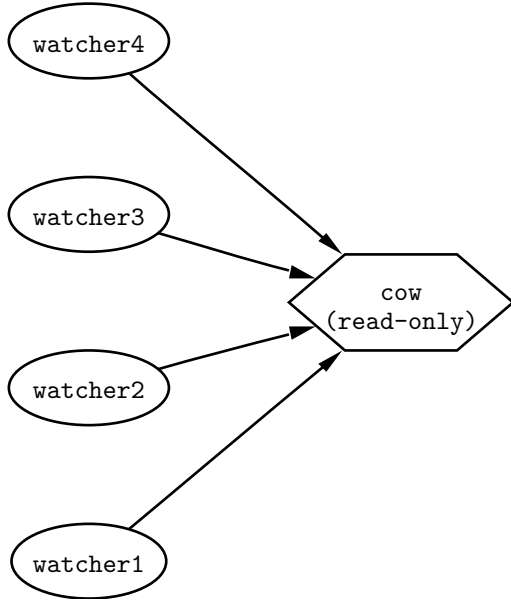
3.1.3 Containers

C++ is not a fully object-oriented language. Its primitive data types (such as `int`) cannot be handled as objects, therefore they cannot be incorporated into the object model without some extra undertaking. The ultimate goal is to be able to handle these primitives as if they were true `GSOjects`. In `GSLib` the issue can be handled by wrapping the primitives in *container* templates as *payload*. Some programming language call this technique *boxing*. The containers are valid `CObjects`: they are reference counted, and can be passed to methods that only accept `CObjects`. In addition, they are making provision for the voluntary *interfaces* of `GSOject`. Containers are crucial for the implementation *collections* in `GSLib`.

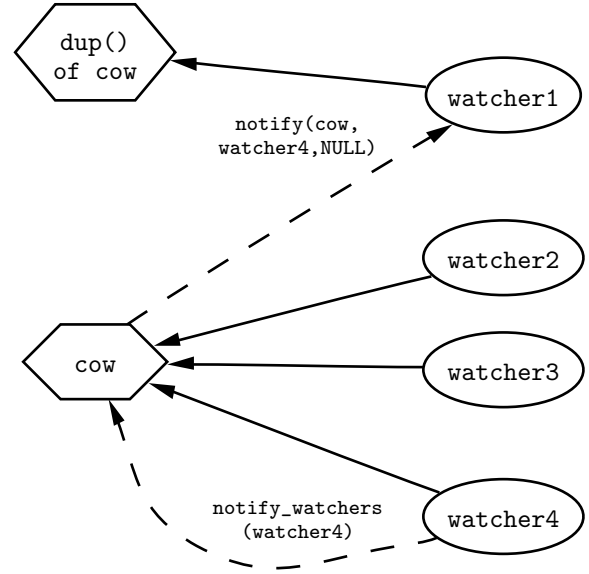
container
templates

Data types are divided into four categories: `Atom`, `Struct`, `CObj`, and `CCow` (not to be confused with the `CCow` class). All of them has a corresponding container template which can be used for wrapping. Categorizing data types was necessary because the containers must know at least that much about the data types they are handling, and C++ has no syntax for selecting between code fragments based on template arguments (much like `#if arg == ... #elif ...`). Container classes are unrelated in the object-oriented sense, because the signatures of their methods vary with the data type the container wraps.

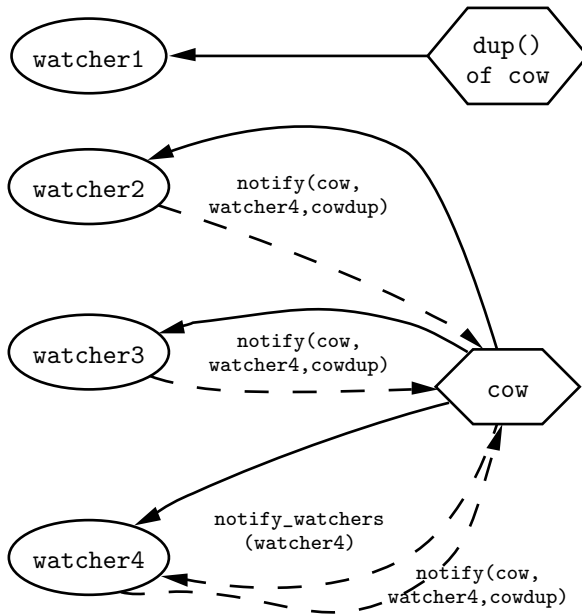




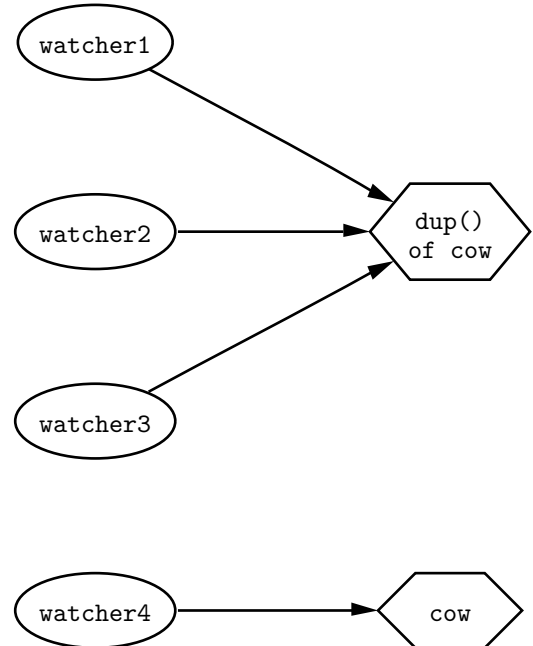
(a) Initial state. `cow` is shared among four watchers. Solid lines represent *has* relationships.



(b) The fourth watcher indicates its desire to have a private copy of `cow`. `cow` begins to notify the watchers about it. The first one `dup()`s `cow`, and `release()`s the original object. Dashed lines represent message routes.



(c) The notification of `cow` reaches all watchers.



(d) Final state. `cow` is owned by the fourth watcher exclusively; the other watchers share the duplicate of `cow`.

Figure 3.2: The process of copying on write

- `GSOBJ_CONTAINER(category, data_type)`¹:

Expands to a sequence instantiating the appropriate container template embedding `data_type`. This is preferred over spelling out the instantiation because it hides the line noise.

- `TContainerAtom<data_type>`:

For types you wish to pass by value when setting/retrieving the payload of the container. Typically, `data_type` will be `char`, `unsigned`, or some pointer.

- `TContainerStruct<data_type>`:

The payload is communicated by address (nevertheless it is stored in the container in its entirety).

Containers wrapping `Atomic` and `Struct`-like types can `compare()` and `print()` themselves (that is, their payload), but this is subject to the availability of properly-typed `compare_it()` and `print_it()` functions, which carry out the actual work. Many functions are supplied for the most primitive types (`unsigned`, `double`, ...), but we could not prepare for uncountably many user-defined types. Thus, for example if you wish to use the `IComparable` interface of a `Struct` container, it will be necessary to extend the `compare_it()` set of functions with a new overload which knows how to deal with the `Struct`.



- `TContainerCObj<data_type>`:

Wraps a `CObject`-derivate in a container. Doing so makes sense when the container *type* is passed as an argument to a template which asks for a container specifically. In cases like this the concrete container type has to be passed, since the container templates have no common abstract container ancestor. `compare()` etc. are relayed to the payload directly, so the container will be comparable *iff* the payload is.



- `TContainerCCow<data_type>`:

A descendent of `TContainerCObj`, so everything written about it apply to this container as well. In addition, the container becomes a watcher of the payload cow, therefore it responds to `notify()` events. Upon its receipt the payload is `dup()`ed and/or replaced according to the `protocol`.

- `CContainerVaradic`:

A container whose type of payload can be changed in run-time. This class is not a template, and is typically used to define container-like fields of non-template classes. Its greatest shortcoming is that it cannot be extended to handle arbitrary payload types.



accessing
the payload

What is left from containers is the question of setting and retrieving the

¹Actually, this macro and the `TContainer{Atom,CObj,CCow}` classes have another argument, which specifies the type of the constant pointer if `data_type` is a pointer.

payload. It is unavoidable that containers of different kinds behave differently, but much trouble was taken to get them ‘do the right thing’ consistently.

- `ctor(payload):`

The payload can be set when the container is being constructed. If the container is constructed via the default constructor, its payload will be initialized to some (category-dependant) null value.

- `null_v():`

Returns the *null value* for the type of the payload. Null values are intended to represent undefined payloads. Unfortunately it is only reliable for CObj and CCow containers (for which the null value is simply NULL).

- `assign(new_payload):`

Replaces the current payload of the container with `new_payload`. For Atomic and Struct containers this method boils down to a C assignment. CObj containers pay attention to `release()` the old payload. Ownership of `new_payload` is taken. CCow containers also start watching it.

- `retrieve():`

Returns the current payload. CObj containers retain the ownership of the payload.

- `replace(old_payload *, new_payload):`

Like `assign()`, but the previous payload is returned, whose ownership is given away.

- `payload:`

Alternatively the payload can be accessed directly. Sometimes this is less cryptic, but it may hurt the interchangeability of containers.

3.2 Logging

GSlib and the upper layers use logging to inform the user about the current activities of the system: what it is doing, why is it doing that, how it has interpreted the input, etc. This information is a significant **development aid**. The goal of the `gsmon` subsystem is to offer structured interfaces to generate log messages, to filter those being important at the moment (per user configuration), and to present them in a consistent way.

`gsmon` has three aspects. *Messages* enter the subsystem through the use of `GSMON_LOG*()` macros. The *monitor* receives the messages, and decides which *output driver* to send them. If configured so, the message may not be delivered anywhere. Output drivers are responsible for formatting the messages and transmitting them to the display. **Figure 3.3** illustrates the idea. Unix administrators may realize similarities to `SYSLOG-NG` (which the author has contributed to).

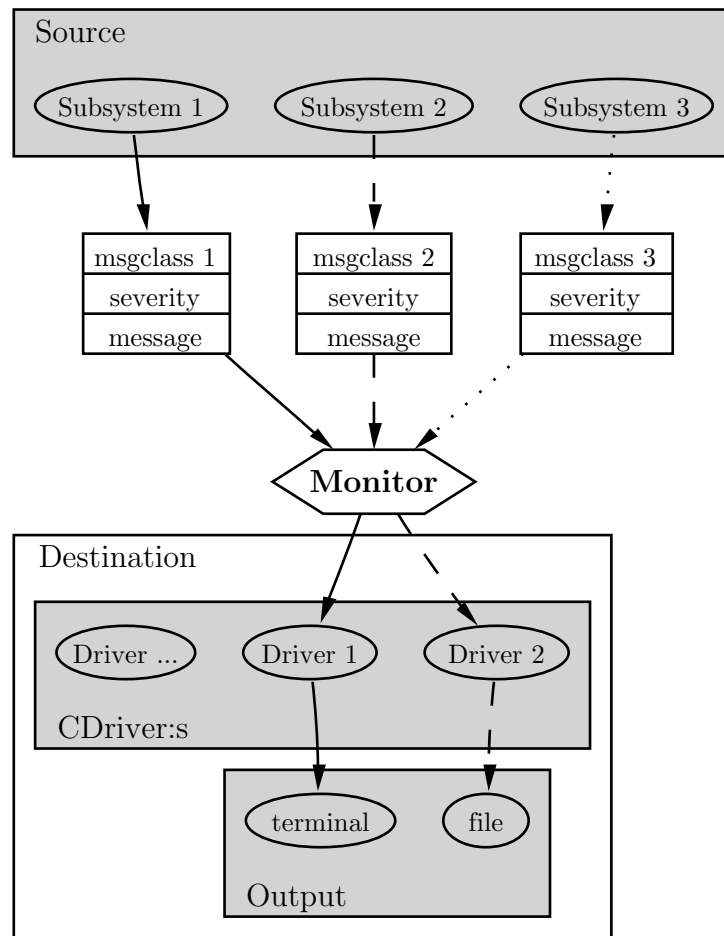


Figure 3.3: Structure of **gsmon**. Differing line styles represent sample message routes. The message of *Subsystem 1* is written to the terminal, while the one of *Subsystem 2* is appended to a file. The third one is eaten up, because the monitor has been configured not to deliver messages of *msgclass 3* at that severity level.

The logging facility can be disabled at compile time by setting `CONFIG_GSDIAG_MONITOR` to zero in the library-wide **configuration header**. Then no messages will even be generated, bringing the process to the end at the very beginning, which may save many CPU cycles if the logging is in a critical path of the application.

messages

Messages are actually composed of several parts beside the actual message text. The first of them is the *message class*. Callers of **gsmon** may group their log messages arbitrarily, for example based on the activity being performed, such as ‘init’ or ‘decode’. The user may configure the monitor to suppress certain classes of messages, reducing output noise. Then, all messages are logged at the specified *severity level* which says how important it is: is it intended for developers, does it warn about a condition that is ought to be investigated, etc. The full list of possible severities are documented in **gsmon/constants.h**. Finally, the monitor is passed the location (source file name, method name and line number) where the message was generated implicitly.

- `GSMON_REGISTER_MSGCLASS(subsystem, message_class, message_class_id)`:

This macro defines a message class for `gsmon`. After that messages of `message_class_id` can be logged, and will be recognized so. The other parameters are strings, used to display the name of the associated entity.

- `GSMON_LOG(severity, message_class_id, string)`:

Log a string message. `message_class_id` must already have been declared with the previous macro. The namespace of `severity` need not be spelled out.

- `GSMON_LOGF(severity, message_class, format, ...)`:

Likewise, except that the message text is given `printf()`-style. This is the way to log numbers, for example.

- `GSMON_ENTER()`:

Logs at `DEBUG` severity that control has entered a function and increases the indentation of further messages. This information makes it easier to deduce the dynamic scope of a log message.

- `GSMON_LEAVE()`:

Undoes the effect of `GSMON_ENTER()`.

drivers Output drivers are C++ classes which know how to display a log message. All of them are derived from `CDriver`¹ (which is a `GObject`). Currently, only one driver, `CDriverStdio` is defined, which writes to `stdio` `FILES`, but in theory one could create drivers for any medium. The drivers are configurable through the public interfaces.

- `setopt_options(mask, enable)`:

Enable or disable the options specified in `mask`, which is a bitmap of `OUTOPT_*` constants. The constants are defined in `gsmon/constants.h`.

- `setopt_fields(mask, enable)`:

Controls what additional information to log along with the message text. `mask` is bitmask of `OUTFLD_*` constants.

- `setopt_format(mask, enable)`:

Controls the appearance of the log messages. `mask` is a bitmask of `OUTFMT_*` constants. At the moment none of the flags is honored.

- `setopt_time_format(time_format)`:

This option is specific to `CDriverStdio`, and sets the exact format of the time field in the displayed log message.

¹All identifiers are in the `gslib::gsmon` namespace.

- `set_tag(tag)`:

Sets the tag for the `OUTFLD_TAG` field.

By default more severe messages (`NOTTICE...FATAL`) are logged to the standard error. Classless functions of `gsmon` are used to tell the subsystem how to route messages.

- `set_route(driver, severities, message_classes)`:

Instructs `gsmon` to send messages whose severity is one of `severities` and whose message class is one of `message_classes` to `driver`. The ownership of `driver` is taken. If `driver` is `NULL`, the route is removed. Returns whether registration was successful.

- `forked()`:

Output drivers cache the PID of the process for the `OUTFLD_PID` field to eliminate frequent calls to `getpid(2)`. When the process `fork()`s, its PID changes, and the cache needs to be invalidated. This is done by this function.

3.3 Error handling

`gserr` The error handling facility of `GSLib` is called `gserr`. It is founded around one of the characteristics of errors that they can be *chained*. A failure of a lower layer may cause failure of its caller, which in turn may make an upper layer fail etc. For the user to fully understand a program failure it is not enough to know the limited information provided by any of the layers; it is the sum of the information which can frame the context. The *error stack* of `gserr` is meant to support this requirement. The stack consists of entries describing errors reported by layers invoked in the latest dynamic path of execution from the least recent one to the most recent. The size of the stack is fixed at compile time, but it can be changed by adjusting `CONFIG_GSDIAG_MAX_ERRORS` in the global [configuration header](#).

- `GSERR_PUSH(message)`:

Pushes a new error report onto the top of the error stack. `message` is the textual error message, intended for humans.

- `GSERR_PUSHF(format, ...)`:

Likewise, but the message is formatted with `printf(3)`. Compare [GSMON_LOG\(\)](#) and [GSMON_LOGF\(\)](#). Both this and the previous macro have a `*_EC()` counterpart, which allow for supplementing the report with an integer error code, which can be used by the receiver to interpret the report programmatically.

- **GSERR_PUSH_SYS(function):**

Like `perror(3)`, but the error message is pushed onto the stack. This macro is intended to report errors returned by functions of the C library.

error propagation

`gserr` does not enforce any scheme for the propagation of the error signal. Both C-style return codes and C++-style exceptions are acceptable.

- **struct error_entry_st:**¹

This structure (defined in `gserr/definitions.h`) holds an error report; in particular, the location where the report was generated, the textual error message, and the (optional) error code.

- **GSLIB_THROW*():**

These are a series of macros defined in `gslib/macros.h` which help you to propagate the error signal via exceptions. Generally, the macros create error reports based on the arguments, push them onto the error stack, and throw an `error_entry_st` pertaining to this error. This structure can be examined in a `catch` block.

- **throwit():**

This function is internal to `gserr`, but it can be very useful during debugging. Every exception thrown by the macros above goes through `throwit()`, exposing a convenient hook which can be breakpointed to pause the program when an error is reported.

Error reporting is just the half of what is needed for proper error handling.

error examination

The error stack can be examined by the following functions.

- **lasterr():**

Returns the most recent error report from the top of the stack.

- **preverr():**

Returns the error preceding `err`. Together with the previous function they can be used to cycle through all the reports from the most recent (generated nearest) to the least recent (generated deepest in the dynamic scope).

- **forget_all():**

Clears the error stack. Must be called explicitly when the error reports are no longer of interest (e.g. the exception handler finished processing them).

- **GSERR_LOG(severity, message_class_id, format, ...):**

Like `GSMON_LOGF()`, but it also dumps the error stack. After that the stack is cleared automatically.

¹All identifiers are in the `gslib::gserr` namespace.

3.4 Collections

The collections of GSLib are very versatile data structures. They try to combine the best breeds of linked lists, hashes, arrays, stacks, and queues. Their GSAvl implementation – which we refer to as GSAvl – is based on templates, which makes it possible to store any types of data in the collections. GSAvl complies with the semantics of the **object model** of the library.

STL The may be raised rightously why not use the Standard Template Library (STL), which comes with the C++ compiler for free. The answer has just been said: the STL does not (can not) fit in our **object model**, its classes do not have the expected interfaces and do not expose expected behavior. Compensating for these deficiencies with thin wrappers seemed a futile attempt.

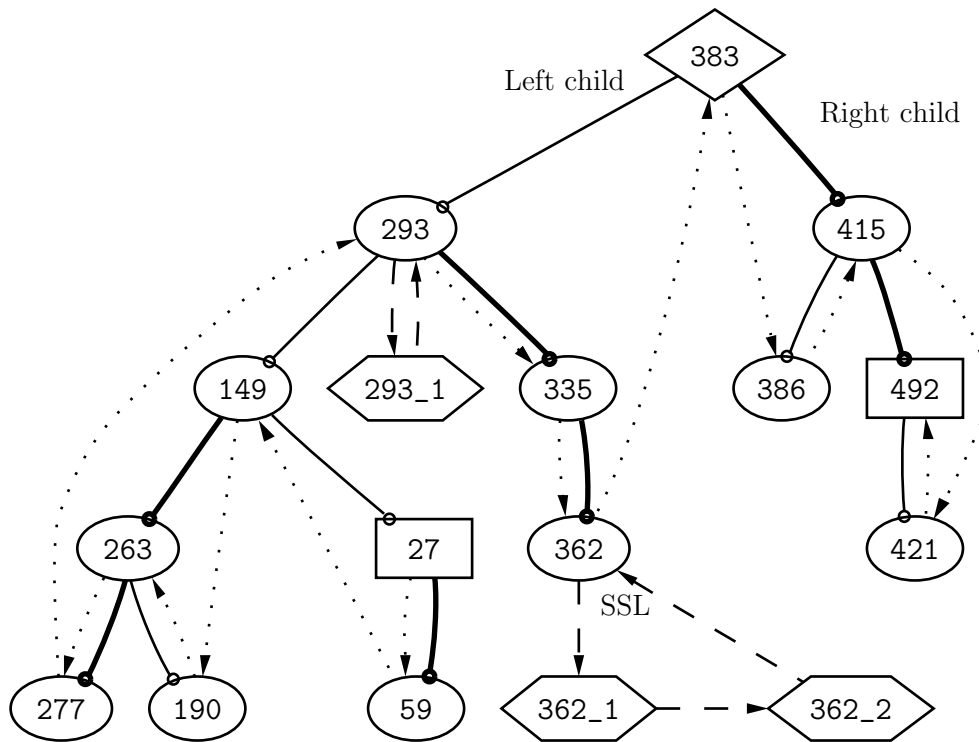
RA One might further argue that the invention of **GObject** was a waste as a whole, since STL has smart pointers already. True as it is, except that **GObject** does much more than reference counting. Besides, the author views the philosophy of the STL (*Resource Acquisition Is Initialization*, RAI) a well-marketed hack to make C++ *appear* as if it had automatic garbage collection. RAI suggests allocating *all* objects on stack, which *are* reclaimed when the control leaves the frame. It is nothing new compared to C, except that C++ takes the responsibility to call the destructors of these objects, so they can clean up after themselves. This *is* automatic and garbage collection. The problem is that dynamically allocated objects do not enjoy this guardianship, so RAI outlaws them as legacy heritage. Of course many things (such as recursive data structures) cannot be done without resorting to pointers, so more workarounds (such as the smart pointers mentioned above) have been given birth to *hide* the problem. This proliferation of kludge is what the author could not take in.

Finally, there is the question of extensibility. Our concern was: if the collections of STL happen to lack some feature whose necessity we cannot decline, is there a way out? **Extreme programming** suggests avoiding commitment to a particular infrastructure unless no other options are feasible, and it does so for good reason: extensible software is expected to have (relatively) **sane and documented** internals, which is often not the case. To illustrate the situation of **LIBSTDC++3**, the then-current version of the standard C++ library implementation for Linux, let us quote a comment from a file (**basic_file.h**) distributed with the library:

Ulrich [the chief **GLIBC** supervisor] is going to make some detailed comment here, explaining all this unpleasantness, providing detailed performance analysis as to why we have to do all this lame vtable hacking instead of a sane, function-based approach. This verbiage will provide a clear and detailed description of the whole object-layout, vtable-swapping, sordid history of this hack.¹

structure CTree Back to GSAvl details, all of its collections are derivatives of the **CTree** class²,

¹The current version (**LIBSTDC++6**, **basic_string.h**) has another amusing interposition in it: “Documentation? What’s that?”. To be fair, this version seems to be much better commented.



CNode which holds together CNode instances, called *primary nodes*. The nodes are arranged in an AVL tree, regardless of whether the actual data structure the collection implements is an array or a hash. The code managing the AVL-tree (e.g. rotation) is an improved version of what was found in [AAPL](#). The nodes are automatically linked into a list (the *primary list*), by which they can be enumerated in order. For this twofoldness, a GSAvl collection can be regarded either as a list or as a tree, and it is common to refer to them by these terms. Primary nodes may head *secondary nodes* (or *siblings*) linked in the *secondary list* (SLL). The SLL is circular, for historical reasons. Secondary nodes can only be retrieved by their primary node; list operators will not normally access them. [Figure 3.4](#) shows the most important relationships.

Collections come in three different flavors. The library user can choose between them based on what she needs to store. **Arrays** are best when the elements only have *values*, and they are to be accessed by index numbers. **Array lists** offer operations to add and remove elements at the head, at the tail, or at any random place, so it can also be thought of either as a stack, as a queue, or as a linked list. One interesting feature of **GSavl Arrays** is that

²Of the `gslib::gsavl` namespace.

indices need not be continuous: there may be an element in the third and in the ninth slot, but nothing between.

OrSet **OrSets** (*ordered sets*) are good fit when the elements shall be looked up by themselves, i.e. when they are the *keys* to themselves. This is handy to encode information as the presence/absence of an element in the **OrSet**. This flavour of lists does not allow for values.

VOrSet **VOrSets** are capable of more in exactly this regard. **VOrSets** can be conceived as the **GSLib** equivalent of Perl hashes (while **Arrays** could be modeled as **VOrSets** whose keys of nodes are the indices of the nodes in the list). All three flavors support the ordered enumeration of their elements (this is why **OrSets** are ‘ordered’).

VaList Two other types of lists are built upon these flavors. They are jointly referred to as **VaLists**, and the speciality about them is that they can do math with their values. For example, it is possible to numerically add matching elements of two lists, or to multiply all values of a list by a scalar. **TVaList** show the characteristics of an **Array**, while **TKaList** is like a **VOrSet**.

declaring collections As suggested earlier, the exact data type of keys and values does not matter to **GSAvl**, apart from that keys must be **Comparable**. Keys and values are embedded in **containers** in the nodes. This makes the concrete type of nodes and lists depend on the type of the keys and values, therefore the appropriate template classes must be instantiated to enter into the possession of a collection. The macros of **GSAvl/macros.h** help in this task. See the header for the full range of possibilities. See **test/simple.cc** for examples.

- **GSAVL_ARRAY**(value_kind, value_type)
- **GSAVL_ORSET**(key_kind, key_type)
- **GSAVL_VORSET**((key_kind, key_type), (value_kind, value_type)):

These macros expand to the C++ type of a list, which can be **typedefed** or used to define a variable. The semantics of the arguments follow that of **GSOBJ_CONTAINER()**. Note the parentheses in the arguments of **GSAVL_VORSET()**; they are *required*.

list-wide operations The heart of **GSAvl** collections lie in the operations one can do about them. The first group of operations affect the list as a whole. Since lists are full-rank **GSOBJs**, the ‘standard’ interfaces are implicitly available as well.

- **configure**(parameter, value):
With this method you can configure the run-time behavior of the list with respect to unanticipated events, such as when a particular node is searched but none found. The defaults are biased toward the C-style ‘check the return’ scheme, but it may not always be the best choice. See **GSAvl/definitions/CTree.h**.

- **size()**:
Returns the number of primary nodes in the list.

- `volume()`:
Returns the number of all nodes (primary + secondary) in the list.
- `trim()`:
Eliminate all secondary nodes from the list.
- `truncate()`:
Empty the list completely.
- `merge(other, depth)`:
Add all nodes of the `other` list to the receiver. In case of `Arrays` `other` is effectively appended to the receiver. For other flavors of lists the result depend on the list configuration; by default nodes whose key does not exist in the receiver are added, otherwise they replace their counterparts. Unless directed otherwise nodes are deep-copied.
- `subtract(other)`:
Remove all nodes whose keys are present in the `other` list. This operation is not defined for `Arrays`. (Rationale: in `Arrays` there are not keys.)
- `intersect(other)`:
Only keep those nodes whose keys are present in the `other` list. This operation is not defined for `Arrays`.

list node operations The second group of operations are which deal with individual nodes, and they are many and numerous. Library users will spend most time with these methods, and may recognize heavy Perl influence. Almost all of the methods take multiple calling conventions; these are indicated in the suffix of the method name. For example, one version of `find()` returns the node found, another returns with its value directly. Here we only list the versions taking or returning nodes; the semantics of the others are hopefully deducible from these descriptions examining their prototypes.

- `node_t`:
The exact C++ type of nodes depend on the type of their keys and values. It is possible to recreate the node types by the `GSAVL_*_NODE()` macros of `GSAVL/macros.h`, but the `node_t` member of list classes has exactly the same meaning.
- `mknode0()`:
Returns a newly allocated node of the appropriate type (`node_t`), but does not add to the list. The key and/or the value of the node are `null`.
- `is_in_list(node)`:
Returns whether `node` is in the list, either as a primary or a secondary node.

- **idx_n(node):**
Returns the number of nodes ranked less than **node** in the primary list. Indexing starts from zero.
- **unshift_n(node):**
Inserts **node** at the beginning of the list. Only meaningful in **Array** context.¹
- **push_n(node):**
Inserts **node** at the end of the list. Only meaningful in **Array** context.
- **pred_nn(where, node):**
Inserts **node** just preceeding **where**. (Think **splice()** of Perl.) Equivalent to **unshift_n()** if **where** is **NULL**. Only meaningful in **Array** context.
- **succ_nn(where, node):**
Inserts **node** just succeeding **where**. Equivalent to **push_n()** if **where** is **NULL**. Only meaningful in **Array** context.
- **enter_n(node, depth):**
Adds **node** to the list at the appropriate place. For **Arrays** it is equivalent to **push_n()**. **depth** observes **copy()** semantics, i.e. ownership of **node** is taken if **depth** is zero, ownership is shared if the parameter is one, otherwise a **dup()** of **node** is added to the list.
- **nth_n(index):**
Returns the node at the **index**th position in the primary list of the collection. Think of it as a subscript operator. If there is no such node the result depends on how the list is **configure()**d.
- **any_n():**
Returns a **rand(3)**only chosen node from the list.
- **find_n(key):**
Finds and returns the node whose key is **key**. Does not search the secondary list of nodes. Only meaningful in **OrSet** and **VOrSet** contexts.
- **unlink_n(node, is_chainwide):**
Removes **node** from the list. Returns **node**, its ownership given away. If the operation **is_chainwide** **node** will keep its siblings (which then cease to be part of the list); otherwise the first sibling of **node** takes its place.

¹In theory this and the following few methods can be used in **OrSet** context as well, given that the key of **node** compares between **where** and its predecessor, otherwise the integrity of the list is ruined.

- `destroy_n(node, is_chainwide)`:

Removes `node` from the list and `release()`s it. If the operation `is_chainwide` siblings of `node` will go away too, otherwise the first one takes its place.

- `shift_n()`:

Unlinks and returns the first node of the list.

- `pop_n()`:

Unlinks and returns the last node of the list.

Nodes are not necessarily part of any list; they can be created standalone to enter a list later, or be detached from one via `unlink_n()`. The following standalone operations are available in either case.

standalone
node operations

- `key, value`:

These fields embed the key and / or value of the node. They are `containers`, so to access the value of a node for example one writes `node->value.payload` or `node->value.retrieve()`.

- `is_primary()`:

Tells whether the node is part of a list and is primary, i.e. `find_n()` can find it.

- `get_head()`:

Returns the primary node in the secondary list of the node. If the receiver `is_primary()` then it returns itself.

- `is_in_list(list)`:

Tells whether the receiver is in `list`, or in any list if the argument is `NULL`.

- `get_list()`:

Returns the object reference to the list the node is part of, or `NULL` if there is no such list.

- `HPrev()`, `HNext()`:

Return the node preceeding / succeeding the receiver in the primary list. Standalone and secondary nodes are not on primary lists, hence the methods return `NULL` for them.

- `VPrev()`, `VNext()`:

Return the previous/next node in the secondary list of the receiver. The secondary list is circular at both ends.

- `has_sibling()`:

Tells whether the node has at least one sibling, i.e. `VNext()` would not return the receiver itself.

- `make_sibling(other)`:

Joins the secondary lists of the receiver and the `other` node such that the former is broken open and pasted between `other` and its left (`VPrev()`) sibling. `src/GSLib/CNode.cc` has a nice diagram showing the exact layout.

- `unmake_sibling()`:

Unlinks the receiver from its secondary list. The node must not be primary; use the `unlink_n()` list operation in that case.

- `kill_siblings()`:

`release()`s all the siblings of the receiver, which must either be primary or not part of any list.

- `dismiss()`:

`release()`s the receiver node, and returns its value.

3.5 XML driver

GSXml The purpose of the XML driver of GSLib is to allow developers to create XML printouts quickly and easily. To achieve that the first task is to build the internal GSXml representation of the XML document. From the library user's perspective this usually is done by invoking the `serialize()` standard `GObject` interface, which returns a GSXml object representing an XML subtree: the receiver and other objects that logically belong to it. It is possible to traverse and manipulate this tree; however GSXml is optimized to extending and finally printing the XML structure to a file stream. This is realized by calling the `print()` standard interface of the returned GSXml object.

The carefree `print_it()` function is provided to automate the whole procedure. It has two accompanying functions, `start_output()` and `end_output()`, which ensure that the output is a valid XML document.

structure The GSXml representation is a very simple, tree-like structure, which describes what XML nodes constitute the document and how they are organized. **Figure 3.5** gives an idea about the high-level structure of the tree. From the driver's point of view an XML document consists of nodes (`<Something>...</Something>`), node attributes (`<Something attr="value"/>`), and freestanding text. They are mapped to `CNode`¹, `CAttr`, and `CLeaf` class of objects respectively, all of them being proper `GObjects`, and proper `GSavl` nodes at the same time.

¹Not to be confused it with `gslib::gsavl::CNode`. All identifiers introduced in this section are located in the `gslib::gsxml` namespace.

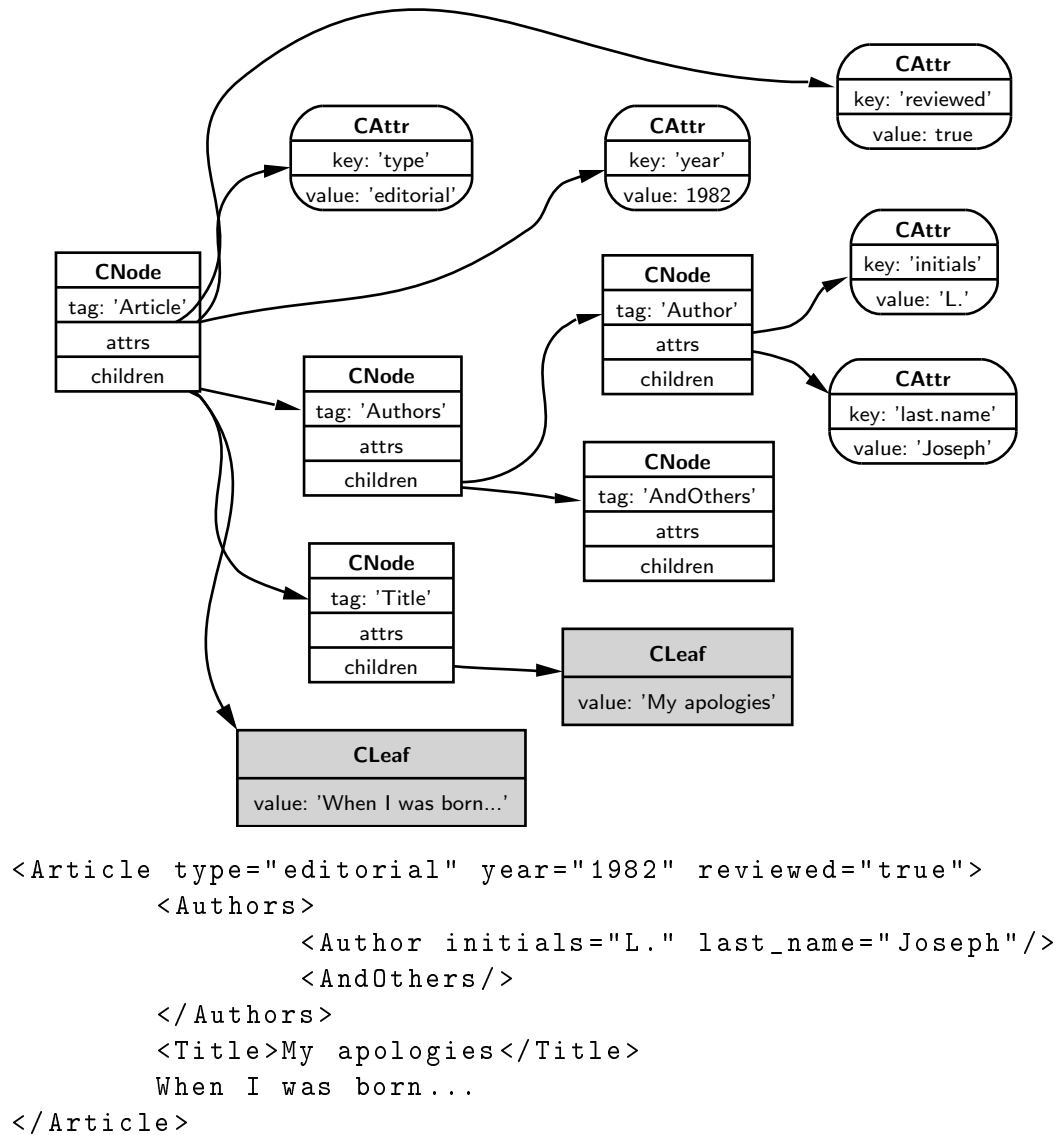


Figure 3.5: The internal GSXml representation of the XML document underneath. (Read left-to-right, top-to-bottom.) Framed drawings symbolize objects of the corresponding class set at the top of the drawings. Below that are the relevant fields of the objects. Arrows denote *has* relationship between objects.

As it should be clear from the figure, the tree is defined recursively. In essence, a `CNode` is a handle for a (sub)tree. A `GSXml` node may have attributes, may contain freestanding text, and may link to other nodes.

- `tag`:

This is the strings what `print()`s between the angle brackets. It is not `free()`d upon the destruction of the node, so it is advisable not to assign it a dynamically allocated string.

- `attrs`:

This is a `OrSet` of `CAttrs`, and it can be manipulated by regular `list-wide operations`. The set is indexed by textual attribute names.

- `children`:

This is an `Array`-like collection, gathering both `CNodes` and `CLeafs` mixed in any order. The children are `print()`ed in the order they are in `children`.

- `find_child_by_tag(tag)`:

Would be `children.find_n(tag)` if it were indexed. But it is not, so this methods searches `children` linearly, and returns the first node whose `tag` matches.

- `ctor(tag)`:

This constructor initializes `tag` to the specified value. Since `tag` must always be set, this is the only way to create a `CNode` object from the scratch.

`CAttr` While `GSXml` attributes are collected in a pure `OrSet`, `CAttrs` *do* have a value—it is only that this fact is concealed from its list to allow for varadic attribute value types.

- `value`:

This is a `varadic container` of the attribute value.

- `ctor(name, value)`:

Creates a `CAttr` object, initializes its key and value. `value` can be of any C++ data type `CContainerVaradic` is ready to handle.

`CLeaf` `CLeafs`, representing freestanding text content in an XML node, are much like `CAttrs`, except that they do not have an indexing key. We note here that textual content only undergo basic encoding when `print()`ing it, i.e. the characters '`<`', '`>`', '`&`', and '`"`' are encoded, but nothing else is (such as 8-bit accented characters). The `charset` argument of `start_output()` may help to preserve the XML-validity of the output.

protocol Developers who wish their object to be serialisable need to implement the `ISerialisable` interface, which consist of just one method:

- `serialize(options, base):`

This method returns a GSXml CNode, suitable to be added to the `children` of another node. If `base` is not `NULL` all contents must be added to that node; this is for the caller in order to control where its objects serialize themselves. Otherwise the base node needs to be created, setting its `tag` appropriately. Then add GSXml attributes and children nodes as best represent the object, possibly by asking subordinate objects to `serialize()` themselves, and adding the returned nodes to the `children` of the `base` node. `options` is a `VOrSet` of strings pointing to unsigned integers, which is intended as a platform free from all encumbrances for communicating directives about what the caller wants to be included in the dump. Beware that `options` can be `NULL`. When serialization is done, return the base node.

3.6 Test suite

Basically, the `second generation` GSLib was raised over its predecessor for the sole demand of `collections`. It may not be surprising then that the test suite of the library has `GSAvl` in focus. but in such a way that the other subsystems are tried just as well with good coverage. The suite is a compilation of standalone test programs, most of them aiming at `streessing` the implementation. The test programs also serve as source of examples on how to use a particular function of the library. The `build system` has `MAKE` targets to run tests, with multiple configurations. It is worth noting that the `Makefile` controls what debugging libraries (such as `ELECTRIC FENCE` or `CCMALLOC`) shall be linked with the test programs. As noted at the discussion of the `developer tools` these libraries carry a performance penalty, which can be unacceptable in long-ranging test runs.

- `simple`: The purpose of this simple test program is to confirm GSLib is functional (i.e. did not break in a fundamental way due to a recent change). The test is successful if the program compiles, links, runs, and prints an XML dump of a list. Looking at the source one can get insight into the basic usage of `GXObject`, `GSAvl`, `GSXml`, and `gserr`.
- `memory`: Allocates a collection and fills it with the specified (large) number of items, displaying statistics about the memory consumption regularly. The use of this program is to learn how well `GSAvl` scales, with references to memory and processor resources.
- `trees`: This is the main test for `GSAvl`. It has two modes of operation: Running on its own it fills a collection with random items, then removes them in the same order. This is not very interesting, but in the early days of development it allowed us to examine the AVL properties of the trees as they were growing.

In the other mode it's playing as the mate of `trees-co.pl` in a `co-testing` intercourse. This case the Perl program commands the mate what to do about the list, both parties perform the operation, then compare the contents of their lists. The exact sequence of commands/events can be reproduced and recorded, greatly adding to debug information.

The `build system` knows about the following test configurations:

- `make treetest`: Test `Arrays` and `VOrSets` thoroughly standalone. Can take hours if `CONFIG_DEBUG` is full.
 - `make qtreetest`: Likewise, but make it shorter.
 - `make cotreetest`: Launch a long test in `co-testing` mode.
 - `make qcotreetest`: Likewise, but keep it short.
 - `make hcotreetest`: Run a series of very long `cotreetests`. We found it useful in isolating deeply hidden memory leaks.
- `compare`: This is the co-testing mate of `compare-co.pl`. Together they examine the `compare()` interface of `GSavl` lists by the comparison of random collections.
 - `make acmptest`: Test `Arrays`.
 - `make scmptest`: Test `OrSets`.
 - `make cmpptest`: Test one after the other.
 - `buckets`: This program demonstrates the use and tests the implementation of `containers` and buckets. (*Buckets* have a structured interface for throwing objects at, and they are supposed to select one of them by some undefined criteria.)

Chapter 4

In closing

We have overviewed the functional structure of **MenuGene**, and examined the basic theory of genetic algorithms as applied to the problem of optimal dietary menu generation. Later on we gained insight into the development environment—the workplace of the developers. Lastly we went through the internals of **GSLib**, which constitutes large part of the framework that the problem solver engine relies on, and is arguably the most difficult to understand amongst the system components.

Acknowledging the fact that **MenuGene** stands on its own, and can generate menus fulfilling its **purpose** we can conclude that genetic algorithms are a feasible method for this kind of problems. It has also shown great potentials, which, however, needs considerable **future** research.

From the software development process we have learnt a number of lessons. First, **rewriting software from scratch** is the luxury of those who possess unrestricted amount of resources. ([17] characterizes it as “the **single** worst strategic mistake that any software company can make”.) It indeed can pay off very well, but the iterative development approach as advocated by **extreme programming** provides much safer grounds for business software manufacturing.

The appropriate choice of programming tools and languages may compensate for these time delays. In our case **PERL** offered an appealing alternative to C++, even in areas (e.g. GUI programming) that are not commonly thought of as the prime examples of the applicability of script languages. Its introduction meant significant boost to the project, visible in the rapid growth in the number of applications, such as **rulez** and **subborn**.

Finally, we have found that it is worth investing time into the development tools. Our **BITS** served us very well in case of trouble, which came inevitably. They have an important role from another aspect as well: these are the home-grown pet programs that have the best chance for living on then the current project is finally closed.

Chapter 5

Appendices

5.1 Bibliography

- [1] *PostScript Language—Tutorial and Cookbook*. Adobe Systems, Inc., 1985.
- [2] C. ALBING and M. SCHWARZ. *JavaTM Application Development on Linux[®]*. Prentice Hall Professional Technical Reference, Boston, 2004.
- [3] K. BECK. *Extreme Programming Explained*. Addison-Wesley Professional, 1999.
- [4] B. CASSELMAN. *Mathematical Illustrations: A Manual of Geometry and PostScript*. University of British Columbia, Vancouver, 2005.
<http://www.math.ubc.ca/~cass/graphics/manual>.
- [5] M. CLINE. C++ FAQ lite, 2007.
<http://www.parashift.com/c++-faq-lite>.
- [6] Á. ENDRŐDI. qbot: A framework for the creation of robots testing network-programswriting. Talk at *Semiannual Symposium on Software Engineering*. Dept. of Information Systems, University of Veszprém, January 2007.
- [7] B. GAÁL. Intelligens menügenerálás táplálkozási tanácsadó rendszerhez. TDK thesis, University of Veszprém, Hungary, 2003.
- [8] *SWEBOK[®]: Guide to the Software Engineering Body of Knowledge*, 2004 edition. IEEE Computer Society.
<http://www.swebok.org>.
- [9] *IEEE Std 1008-1987: Standard for Software Unit Testing*. IEEE Computer Society, 1986.
- [10] I. JOYNER. C++?? : A critique of C++, 3rd edition, 1996.
<http://burks.brighton.ac.uk/burks/pcinfo/progdocs/cppcrit>.
- [11] D. E. KNUTH. *The T_EXbook*. Addison-Wesley, 1984.
- [12] D. E. KNUTH. A torture test for T_EX, version 3. Technical report, Stanford University, January 1990.
- [13] B. LINDSAY. Designing for failure may be the key to success. Interview by Steve Bourne. *ACM Queue*, 2(8), November 2004.
- [14] S. MAMONE. The IEEE standard for software maintenance. *ACM SIGSOFT Software Engineering*, 19(1), 1994.
- [15] *CORBA C++ Language Mapping Specification*, version 1.1. Object Management Group, Inc., 2003.
- [16] E. S. RAYMOND. *The Art of Unix Programming*. Addison-Wesley Professional, 2003.

- [17] J. SPOLSKY. Things you should never do, part i, April 2000.
<http://www.joelonsoftware.com>.
- [18] R. M. STALLMAN and THE GCC DEVELOPER COMMUNITY. *Using the GNU Compiler Collection*. Free Software Foundation, 2007.
<http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc>.
- [19] H. SUTTER. Guru of the week. Electronic archive, 1997–2003.
<http://www.gotw.ca/gotw>.
- [20] *National Nutrient Database for Standard Reference, Release 19*. U.S. Department of Agriculture, August 2006.
<http://www.ars.usda.gov/Services/docs.htm?docid=8964>.
- [21] VANDERAJ. PHP top 5. Technical report, Open Web Application Security Project, June 2006.
http://www.owasp.org/index.php/PHP_Top_5.
- [22] F. VON LEITNER. Writing small and fast software. Talk at *Chemnitzer Linux Tag 4*, January 2001.
- [23] D. A. WHEELER. Secure programming for Linux and Unix HOWTO, 2003.
<http://www.dwheeler.com/secure-programs>.
- [24] Commercial company compiles kernel with ICC. KernelTrap forum thread.
<http://kerneltrap.org/node/3866>.

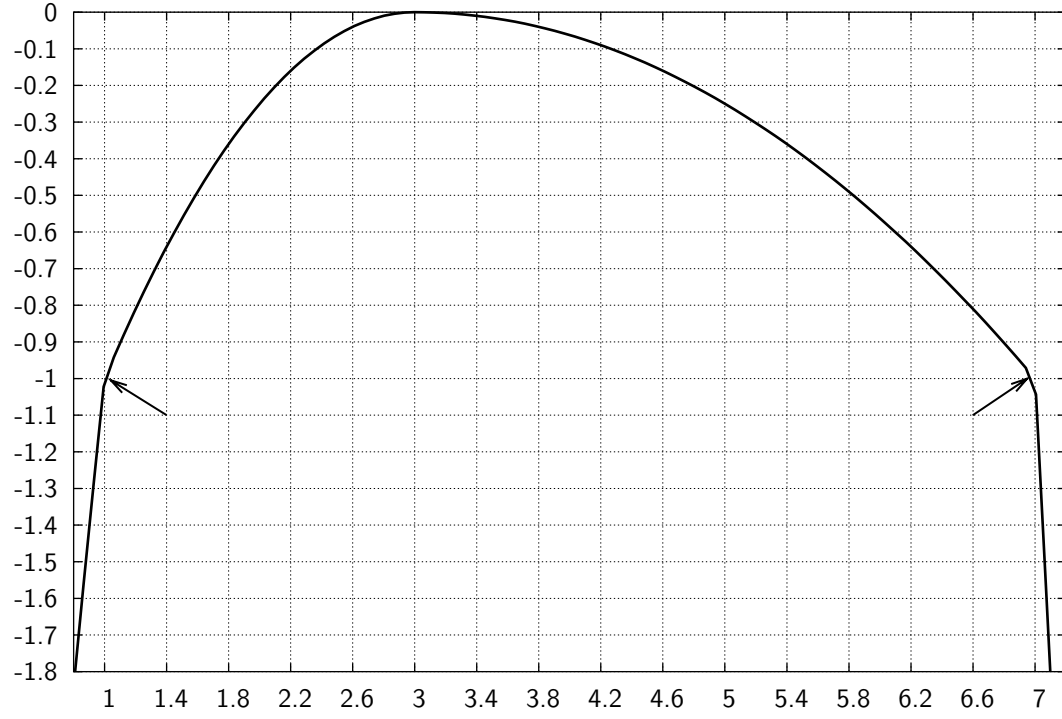
5.2 Progliography

Aapl	C++ Template Library
→ pp. 16, 17, 51	http://www.sourceforge.org/Programming/Libraries/Utilities/aapldev-2.14.tar.gz
Bugzilla	Web-based general-purpose bugtracker
→ p. 28	http://www.bugzilla.org
ccmalloc	C/C++ memory profiler and memory leak detector
→ pp. 28, 31, 60	http://www.inf.ethz.ch/personal/biere/projects/ccmalloc
cons	PERL-based software construction tool
→ p. 26	http://www.dsmiit.com/cons
Cordelia	Szív-érrendszeri kockázatmerő és tanácsadó oldal
→ pp. 6, 16, 17, 27	http://cordelia.vein.hu
Exuberant Ctags	Multilanguage reimplementaion of the Unix ctags program
→ pp. 27, 35	http://ctags.sourceforge.net
CVS	The Concurrent Versioning System
→ pp. 26, 27	http://www.nongnu.org/cvs
darcs	David's Advanced Revision Control System
→ p. 4	http://www.abridgegame.org/darcs
Date::Parse	PERL module for parsing textual time expressions
→ p. 34	http://search.cpan.org/~gbarr/TimeDate-1.16/lib/Date/Parse.pm
DBD::Oracle	ORACLE driver for the PERL Database Interface module
→ p. 34	http://search.cpan.org/author/PYTHIAN/DBD-Oracle-1.19
doxygen	JAVADOC-like documentation system for C, C++ and other languages
→ p. 28	http://www.doxygen.org
Eclipse	An open development platform
→ pp. 27, 29, 35	http://eclipse.org
Electric Fence	A malloc() buffer-overrun debugger that uses the VM hardware
→ pp. 28, 29, 60	http://www.pf-lug.de/projekte/haya/efence.php
GALib	Matthew's genetic algorithm library
→ pp. 9, 15, 29	http://lancet.mit.edu/ga
gcc	The GNU Compiler Collection
→ pp. 25, 26, 33, 35, 37	http://gcc.gnu.org
gdb	The GNU Debugger
→ pp. 27, 31	http://www.gnu.org/software/gdb/gdb.html
glibc	The GNU C Library
→ p. 51	http://www.gnu.org/software/libc/libc.html
gobjc	Objective-C frontend of GCC
→ p. 34	http://gcc.gnu.org
Graphviz	Set of graph drawing tools and libraries
→ pp. 4, 34	http://www.graphviz.org
GTK+	The GIMP Toolkit, version 1.2
→ p. 34	http://www.gtk.org

Gtk	PERL bindings to GTK+
→ p. 34	http://www.gtkperl.org
HTML::Parser	HTML parser class for PERL
→ p. 34	http://search.cpan.org/~gaas/HTML-Parser-3.56
HTML::Tree	Suite of PERL modules for making parse trees out of HTML source
→ p. 34	http://search.cpan.org/~petek/HTML-Tree-3.23
ICC	Intel Compilers for Linux
→ p. 25	http://www.intel.com/cd/software/products/asmo-na/eng/compilers/277618.htm
jam	A powerful, multi-platform MAKE replacement
→ p. 26	http://www.perforce.com/jam/jam.html
Javadoc	Tool for generating API documentation in HTML format from doc comments
→ pp. 28 , 65	http://java.sun.com/j2se/javadoc/index.jsp
JBoss	JBoss an enterprise JavaBeans application server
→ p. 11	http://www.jboss.org
libstdc++3	The GNU Standard C++ Library, version 3
→ p. 51	ftp://ftp.gwdg.de/pub/misc/gcc/libstdc++/old-releases/libstdc++-3.0.tar.gz
libstdc++6	The GNU Standard C++ Library, version 6
→ p. 51	http://gcc.gnu.org
GNU make	Controls the generation of executables and other non-source files
→ pp. 26 , 29 , 33 , 35 , 36 , 60 , 65	http://www.gnu.org/software/make/make.html
Math::Combinatorics	Perform combinations and permutations on lists
→ p. 34	http://search.cpan.org/~allenday/Math-Combinatorics-0.09
Math::Random	Random Number Generators for PERL
→ p. 34	http://search.cpan.org/~grommel/Math-Random-0.69
Monotone	Distributed version control system
→ p. 27	http://monotone.ca
OCI	ORACLE database client tools and libraries
→ pp. 33 , 35 , 36	http://www.oracle.com/technology/software/products/database/xe/htdocs/102xelinsoft.html
Oracle9i	Object-relational database manager
→ pp. 17 , 27 , 34 , 65	http://www.oracle.com/education/description_oracle9i.html
Perl	High-level programming language with an eclectic heritage
→ pp. 33 , 62 , 65	http://www.perl.net
PostgreSQL	Powerful, open source relational database system
→ pp. 17 , 27 , 28 , 31	http://postgresql.org
Prima	Extensible PERL toolkit for multi-platform GUI development
→ pp. 30 , 33	http://www.prima.eu.org
qbot	The IRT test robot
→ p. 28	http://cvs.irt.vein.hu/qbot

QB-ÉLELEM → pp. 9, 11, 17	Digital recipe catalogue by Quadro Byte Zrt. http://gportal.hu/gindex.php?pg=1976400&nid=442991
Subversion → pp. 27, 32	A version control system created to supersede CVS http://subversion.tigris.org
syslog-ng → p. 46	A portable syslogd replacement with enhanced, flexible configuration scheme http://www.balabit.com/products/syslog-ng
TEX → pp. 4, 20	Powerful typesetting programming language http://tug.org
TinyXml → pp. 16, 17	A simple, small, and minimal C++ XML parser http://www.grinninglizard.com/tinyxml/index.html
valgrind → pp. 29, 31	Tool that helps find memory management problems http://www.valgrind.org
Vim → pp. 27, 32	An almost fully-compatible version of the Unix editor vi http://www.vim.org
zsh → p. 34	A powerful Unix shell http://www.zsh.org

5.3 Supplementary figures



$$\text{fitness}(cset) = \sum_{cint \in cset} \min(0, \text{fit}(cint))$$

$$\text{fit}(cint) = \begin{cases} -(cint.o - cint.v)^2 + (cint.o - cint.l)^2 - 1 & | \text{ } cint.v \leq cint.l \\ -\frac{(cint.o - cint.v)^2}{(cint.o - cint.l)^2} & | \text{ } cint.v \in (cint.l; cint.o) \\ 0 & | \text{ } cint.v = cint.o \\ -\frac{(cint.o - cint.v)^2}{(cint.o - cint.r)^2} & | \text{ } cint.v \in (cint.o; cint.r) \\ -(cint.o - cint.v)^2 + (cint.o - cint.r)^2 - 1 & | \text{ } cint.v \geq cint.r \end{cases}$$

Figure 5.1: The fitness function of MenuGene. [7, pp. 61–64] $cset$ is a set of constraints ($cint$). $cint.l$, $cint.o$, and $cint.r$ are parameters of the curve (the accepted minimum, the optimal, and the tolerated maximum amount of a nutrition component in the solution being measured), while $cint.v$ is the actual amount of the corresponding component. The graph depicts an example one-dimensional curve parametrized with $\{1, 3, 7\}$. This function, in fact, *penalizes* deviations from the optimum. (Our implementation also negates the fitness score at the end, but it does not invalidate the point.) At the bounds (pointed to by arrows) the penalty jumps very high, making it infeasible that the solution will compare well with other solutions.

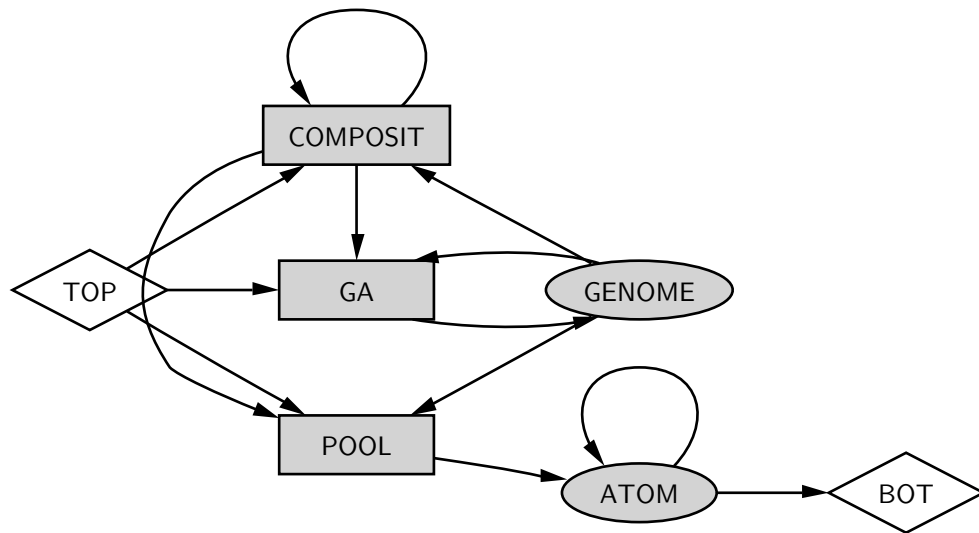
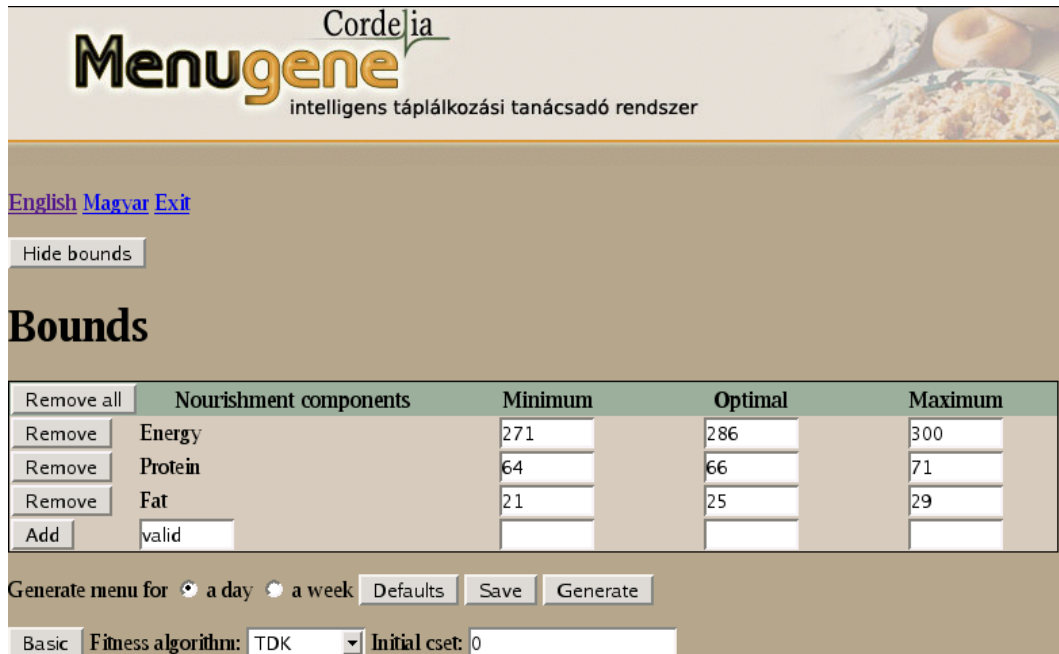


Figure 5.2: Possible relations between solutions and attributes. Boxes enclose attribute types, ovals do solution types. **ATOMic** solutions are ones that are not evolved in a GA population, while **GENOME** ones are. Entities linked with “TOP” are eligible for being at the top of the **GA hierarchy**; similarly for “BOT”.

5.4 Screenshots



Cordelia Menugene
intelligens táplálkozási tanácsadó rendszer

[English](#) [Magyar](#) [Exit](#)

Hide bounds

Bounds

Remove all	Nourishment components	Minimum	Optimal	Maximum
Remove	Energy	271	286	300
Remove	Protein	64	66	71
Remove	Fat	21	25	29
Add	valid			

Generate menu for ☒ a day ☐ a week

Basic Fitness algorithm: TDK Initial cset: 0

(a) Consumer requirements entry

Your daily plan

		Energy	Protein	Fat
Topping	Racos sertés szelet	43	23	6
Garnish	Szekelykaposzta	58	26	7
Soup	Spargakremleves	159	1	1
Dessert	Málnás dessert 100 db	18	15	3
Drink	Gyümölcsivole 5/L	1	--	--
Daily total		279	65	17

Generation took 0:29.5 seconds.

Accuracy		
Nourishment component	Distance	Accuracy
Energy	7	78.221954%
Protein	1	75%
Fat	8	-300%
Total	10.677078	-48.926014%

(b) Generated menu display

Figure 5.3: Screenshots of **junkie**. Design is due to Balázs Gaál.

5.4. SCREENSHOTS

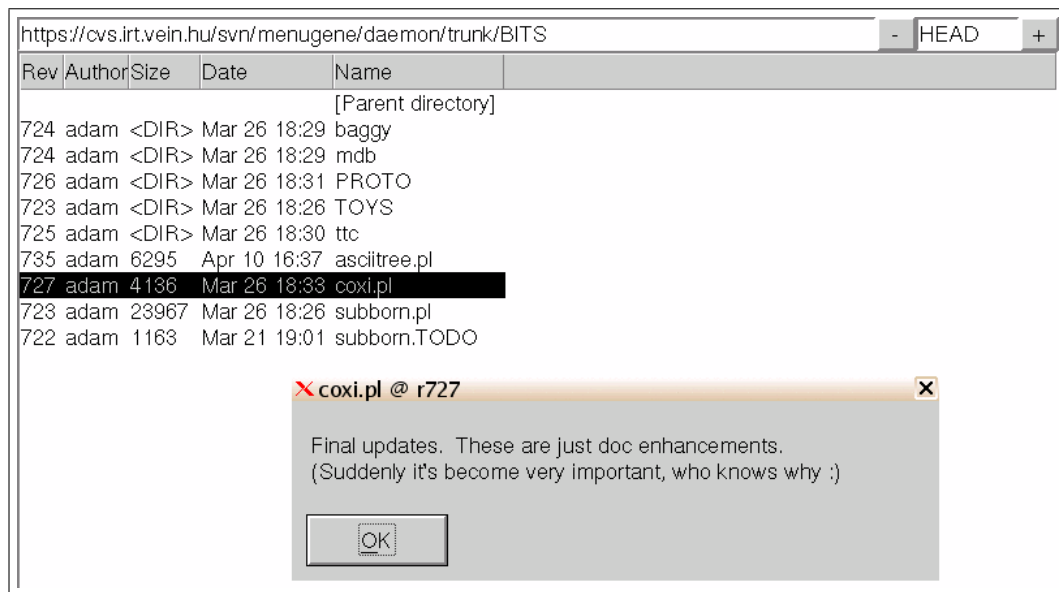


Figure 5.4: Screenshot of `subborn`

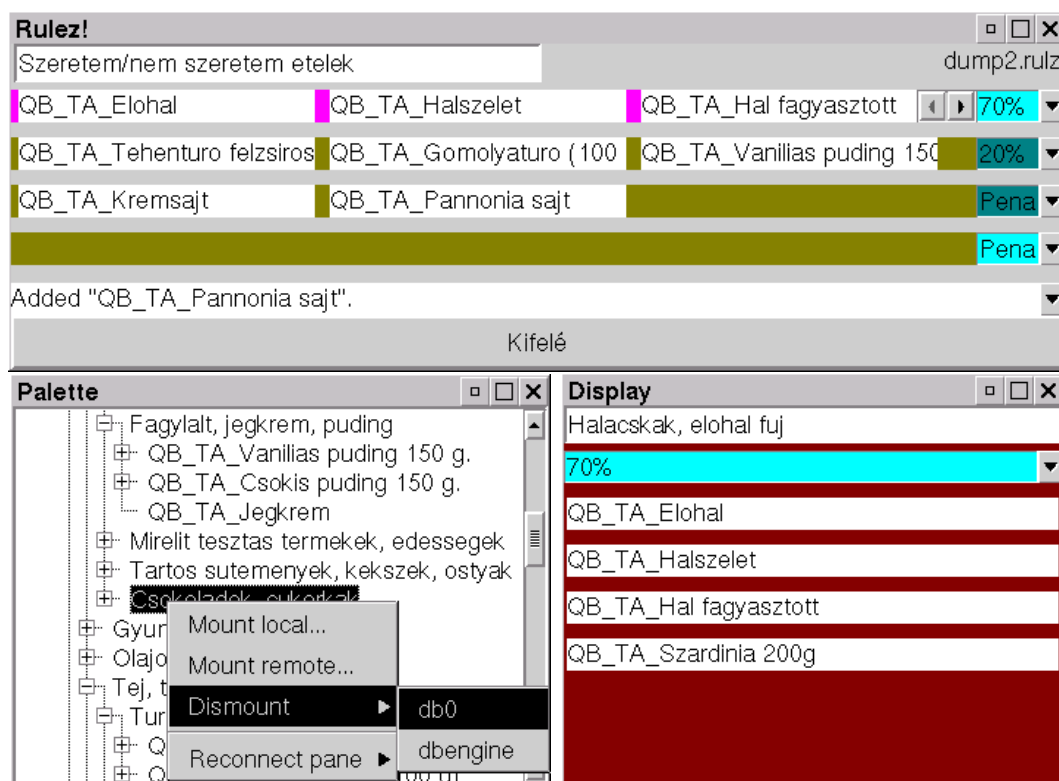


Figure 5.5: Assembling favored and disliked lists of dishes with `rulez`

5.5 Annotated database schema

```
---
--- postgre.sql — draft PostgreSQL schemata of Menugene
---
--- Since the current Oracle DB is not documented anywhere, the best chance
--- you can take is looking at SCHEMA menugene_am, which is close enough on
--- the conceptual level at least to what can be seen in Oracle today.
---
--- Design Recommendations {{{
--- =====
---
--- 1. Choose and follow a naming convention.
--- -----
---
--- The point is to make identifiers recognisable out of context,
--- so you and your friends will have easier time talking about them.
--- You may wish to disambiguate entity tables, relation tables,
--- views, types, functions and such.
--- Also, it is easier to JOIN ... USING rather than JOIN ... ON,
--- so consider calling foo "foo" everywhere in your schema.
---
--- In the meantime it is a good idea to choose your identifiers
--- not to clash with SQL keywords and to leave them case insensitive
--- because otherwise you'll need to quote them all the time.
---
--- 2. Forget NUMERIC. Use INTEGER and unsigned.
--- -----
---
--- NUMERIC is a string that Postgres can calculate with. up to
--- infinite precision. Most of the time you don't need that,
--- so why not use a natural INTEGER? Or unsigned or unreal
--- (see the public schema) for columns you know they won't
--- (therefore shouldn't) contain negative numbers. Well-known
--- candidates are identifier and quantity columns.
---
--- 3. Forget VARCHAR(x). Use VARCHAR and TEXT.
--- -----
---
--- There is no performance penalty imposed by VARCHAR. Neither by TEXT.
--- All that the (x) restriction brings to you is, well, the restriction
--- of max length of the column, which can bite you in the ass very well.
--- Actually, CHAR(n) is worse than VARCHAR because the former will
--- *always* take up overhead + n byte storage due to space padding.
---
--- 4. Use DOMAIN:s to create enumerated types.
--- -----
---
--- The situation is that you have a small set of predefined meanings
```

— and you want to ensure some column can only have one of these.
 — How these meanings encoded is unimportant.

— The textbook approach is to create a table of code–description pairs
 — and have columns REFERENCES to .code. This works, but the price is
 — a new table. Lots of enums, lots of tables, great annoyance keeping
 — them out of way and out of mind. Observe the .description column:
 — it's unnecessary for the DBM and for the application because they
 — can't understand it anyway. It's documentation—for humans.

— PostgreSQL provides an easy way to create new types either restricting
 — or extending old ones. Then you can COMMENT on them to explain the
 — meaning of the codes. Example:

```
— CREATE DOMAIN personality AS "char" CHECK (VALUE BETWEEN 0 AND 3);
— COMMENT ON DOMAIN personality
—     IS '0: nasty, 1: stupid, 2: idiot, 3: crazy';
```

— (You may substitute "char" with SMALLINT or INTEGER.)
 — Executing these commands you will be able to use personality as a
 — first class database object type, and you will have saved one table.

— 5. Use column and table constraints generously.

— Of course, this is what they taught you in database theory classes
 — (or what you tell in those classes if you happen to be the teacher).
 — The usual reasoning is to ensure consistency by making it impossible
 — to enter bogus rows. Another reason is documentation. Whenever
 — a human sees REFERENCES it will immediately become obvious that
 — this table has something to do with another one, without requiring
 — him to read your (hopefully extensive) COMMENTS.

— When you define a column it might be inspiring to ask yourself;
 — — Is there any reason to have two rows in this table sharing
 — the same value for this column? If not, then let it be UNIQUE.
 — — Would it be meaningful to have .column IS NULL?
 — — Can the value of this column be negative under any circumstances?
 — Can we restrict the possible values? Then CHECK().

— It is also wise to require the UNIQUE:ness of relations between
 — entities (UNIQUE (entity1, entity2)), because that's what the
 — application expects most of the time. Never mind the performance cost.

— 6. Learn and use PostgreSQL extensions.

— Some would argue against this due to ill concerns about portability,
 — and advise that you stick with the most widespread features.

```

---
--- There are numerous problems with such views:
--- -- The "common estate" is indeed small. Not even the most basic
--- SQL operations (SELECT, INSERT) are the same across different
--- database managers, not to mention the available data types.
--- Granted, "SELECT * FROM tbl" works the same way everywhere,
--- but RIGHT OUTER JOIN might not. And we haven't talked about
--- transaction isolation, locking, performance etc. which are not
--- just syntactic differences, but behavioral ones. Insisting on
--- the common wealth sacrifices so much that your task may become
--- impossible to carry out. Extensions are made for you, use them.
--- -- The term "portable" is often misunderstood as the condition
--- "I can take *my product* and snap to another standard-conformant
--- component without much hassle". In the SQL realm, portability
--- tends to refer to your ideas and intelligence rather than to
--- your product, which means you don't have to learn a new language
--- from the scratch every time you sit down in front of another DBM
--- because the basic concepts are the same everywhere; you may reuse
--- your previous ideas and experience, but you should not expect it
--- to work the same way as the other one does.
--- -- Database managers are large and complex beasts, and they happen
--- to integrate tighter and tighter with the application every year.
--- Developing in the constant fear of replacing the DBM is like
--- expecting to rewrite your entire product in another programming
--- language. It might happen, yes, but you gotta have strong reason
--- to actually *design* for that.
---
--- While one may disagree with (2) and (3) rebuttal of (1) seems hard.
--- Anyway, my favourite pets are DOMAIN:s, table inheritance, partial
--- indices, indices over expressions, arrays, compound types and EXECUTE.
---
--- 7. Practice your language skills. Document.
--- -----
---
--- Ask your favorite carrier manager and I bet he'll give long lectures
--- about the importance of verbal intelligence. Also think about the
--- many insults you'll be awarded by me for not doing item 2.
--- }}}
---
----- public ----- {{{
---
--- Public harmless stuff that might be of interest for any SCHEMA.
--- Consider adding SCHEMA public to your search_path so you can
--- refer to the goodies here without extra handwork.
---
--- [Maintainer: adam]
CREATE SCHEMA public;
SET search_path TO public;

```


5.5. ANNOTATED DATABASE SCHEMA

```
----- Types -----
-----

-- Use this type (domain) whenever you would use an "unsigned"
-- in a C program, such as declaring object identifier.
CREATE DOMAIN unsigned AS INTEGER CHECK (VALUE >= 0);

-- Unsigned float. One usage area is to express quantities,
-- which cannot be negative.
--
-- To my surprise no programming language known to me offers
-- such data type. (Java is a prime sinner because it does
-- neither have unsigned integers.)
CREATE DOMAIN unreal AS REAL CHECK (VALUE >= 0);

----- public ----- }}}
----- usda_am ----- {{{

-- The usda_am schema is the same as SCHEMA usda, just tidied up a bit.
-- It is intended to be drop-in compatible with the original one.
--
-- Differences:
-- -- Column names are case insensitive.
--   Manual querying was difficult because object names
--   needed to be quoted all the time.
-- -- Columns are defined with more appropriate types.
--   -- INTEGER      => unsigned
--     Usually you mean that and don't want negatives.
--   -- VARCHAR(x)   => VARCHAR
--     Restricting VARCHAR does not buy you anything
--     in PostgreSQL.
--   -- DOUBLE PRECISION => unreal
--     Usually you don't need the extra precision;
--     OTOH you win space and therefore performance.
--   -- .NDB_No has become unsigned
--     Can't quite understand why it was VARCHAR().
-- -- Added reasonable column constraints.
--   -- .*_Desc become UNIQUE.
--     Unfortunately, it broke FUNCTION uj_alapanyag().
--     (FWIW (1) and (2) broke it anyway.)
-- -- TABLE abbrev_menugene INHERITS from TABLE abbrev.
--   Makes more sense: table inheritance is just for that.
--   At least we can get rid of those horrid VIEW:s.
--
-- I recommend that you replace SCHEMA usda by this one.
--
-- [Maintainer: adam]
CREATE SCHEMA usda_am;
```

5.5. ANNOTATED DATABASE SCHEMA

```
SET search_path TO usda_am, public;

--- Functions {{{
-----

-- Finds an unused NDB_No for a new alapanyag, and adds it
-- to food_des and abbrev_menusene. Updates recept.alapanyag.usda_ndb
-- for alapanyag $1. Returns the new NDB_No.
--
-- (Hey V.I., ever occurred to you that you carry the name
-- of a very advanced text editor?)
CREATE FUNCTION uj_alapanyag(unsigned) RETURNS unsigned AS
$$
DECLARE
    recept_alapanyag_id    ALIAS FOR $1;
    usda_id                unsigned;
BEGIN
    SELECT INTO usda_id (MAX(NDB_No) + 1) FROM food_des;

    INSERT INTO food_des (NDB_No, Long_Desc) VALUES (usda_id, NULL);
    INSERT INTO abbrev_menusene (NDB_No) VALUES (usda_id);
    UPDATE recept.alapanyag SET usda_ndb_no=usda_id
        WHERE sorszam = recept_alapanyag_id;

    RETURN usda_id;
END
$$ LANGUAGE PLPGSQL;
--- Functions }}}

--- Tables {{{
-----

-- Food descriptions.
--
-- The following corrections have been made to the table:
--
-- UPDATE food_des SET Shrt_Desc = 'SOY MILK,CHOC,FLUID'
--     WHERE NDB_No = 16166;
-- UPDATE food_des SET Shrt_Desc = 'INF FORMULA, MEAD JOHNSON, '
--     || 'ENF,LAF LIP, W/IR LIQ CON NOT RE'
--     WHERE NDB_No = 3830;
--
-- [TODO] Food groups (.FdGrp_Cd) may be worth importing from USDA.
CREATE TABLE food_des (
    NDB_No                unsigned                PRIMARY KEY,
    -- 5-digit Nutrient Databank number that
    -- uniquely identifies a food item.
    FdGrp_Cd              CHAR(4),
    -- 4-digit code indicating food group to which
```

5.5. ANNOTATED DATABASE SCHEMA

```
Long_Desc      VARCHAR(255) UNIQUE,
               -- a food item belongs.
               -- Long description of the food item.
Shrt_Desc      VARCHAR(60) UNIQUE,
               -- Abbreviated description of the food item.
               -- Generated from the long description using
               -- abbreviations in appendix A. If the short
               -- description was longer than 60 characters,
               -- additional abbreviations were made.
ComName        VARCHAR(255),
               -- Other names commonly used to describe a food,
               -- including local or regional names for various
               -- foods, for example, "soda" or "pop" for
               -- "carbonated beverages".
ManufacName    VARCHAR(255),
               -- Indicates the company that manufactured the
               -- product, when appropriate.
Survey         VARCHAR(255),
               -- Indicates if the food item is used in the USDA
               -- Food and Nutrient Database for Dietary Studies
               -- (FNDDS) and has a complete nutrient profile for
               -- a specified set of nutrients.
Ref_Desc       VARCHAR(255),
               -- Description of inedible parts of a food item
               -- (refuse), such as seeds or bone.
Refuse         unsigned integer,
               -- Amount of inedible material (for example,
               -- seeds, bone, and skin) for applicable foods,
               -- expressed as a percentage of the total weight
               -- of the item as purchased, and they are used
               -- to compute the weight of the edible portion.
SciName        VARCHAR(255),
               -- Scientific name of the food item. Given for the
               -- least processed form of the food (usually raw),
               -- if applicable.
N_Factor       unreal,
               -- Factor for converting nitrogen to protein.
Pro_Factor     unreal,
               -- Factor for calculating calories from protein.
Fat_Factor     unreal,
               -- Factor for calculating calories from fat.
CHO_Factor     unreal,
               -- Factor for calculating calories from carbohydrate.
);

-- This is the abbreviated table, which contains all the food items,
-- but fewer nutrients and other related information than what is
-- provided by all USDA files. It excludes values for starch, fluoride,
-- total choline and betaine, added vitamin E and added vitamin B12,
```

5.5. ANNOTATED DATABASE SCHEMA

```

-- alcohol, caffeine, theobromine, vitamin D, phytosterols, or individual
-- amino acids, fatty acids, and sugars.
--
-- NULL:s for nutrition components mean missing values.
-- .GmWt_* are NULL for some hundred entries, which I find odd.
--
-- Calculate the amount of nutrient in edible portion of 1 pound
-- as purchased by:
--      Y := V*4.536*[(100-R)/100] == (V/100 * 1000) * (100-R)/100 * 0.4536
--      V == nutrient value per 100g (abbrev.{Water,Vitamin_C,...})
--      R == percent refuse (food_des.Refuse)
--
-- (We mean avoirdupois pounds here, which equals to 0.453592 kg.)
-- (NOTE 'V' is in mg/100g for some nutrients!)
--
-- Calculate the nutrient content per household measure by:
--      N := (V*W)/100 == V/100 * W
--      V == nutrient value per 100g (abbrev.{Water,Vitamin_C,...})
--      W == g weight of portion (abbrev.GmWt[12])
--
-- [TODO] Shouldn't we include values for alcohol and caffeine?
CREATE TABLE abbrev (
    NDB_No          unsigned          PRIMARY KEY
                   REFERENCES food_des(NDB_No) ,
    -- 5-digit Nutrient Databank number.
    Water           unreal ,
    -- Water [g/100 g]
    Energ_Kcal      unsigned ,
    -- Food energy [kcal/100 g]
    Protein         unreal ,
    -- Protein [g/100 g]
    Lipid_Tot       unreal ,
    -- Total lipid (fat) [g/100 g]
    Ash             unreal ,
    -- Ash [g/100 g]
    Carbohydrt      unreal ,
    -- Carbohydrate, by difference [g/100 g]
    Fiber_TD        unreal ,
    -- Total dietary fiber [g/100 g]
    Sugar_Tot       unreal ,
    -- Total sugars [g/100 g]
    Calcium         unsigned ,
    -- Calcium [mg/100 g]
    Iron            unreal ,
    -- Iron [mg/100 g]
    Magnesium       unsigned ,
    -- Magnesium [mg/100 g]
    Phosphorus      unsigned ,
    -- Phosphorus [mg/100 g]

```

5.5. ANNOTATED DATABASE SCHEMA

Potassium	unsigned , — <i>Potassium</i> [mg/100 g]
Sodium	unsigned , — <i>Sodium</i> [mg/100 g]
Zinc	unreal , — <i>Zinc</i> [mg/100 g]
Copper	unreal , — <i>Copper</i> [mg/100 g]
Manganese	unreal , — <i>Manganese</i> [mg/100 g]
Selenium	unreal , — <i>Selenium</i> [g/100 g]
Vit_C	unreal , — <i>Vitamin C</i> [mg/100 g]
Thiamin	unreal , — <i>Thiamin</i> [mg/100 g]
Riboflavin	unreal , — <i>Riboflavin</i> [mg/100 g]
Niacin	unreal , — <i>Niacin</i> [mg/100 g]
Panto_acid	unreal , — <i>Pantothenic acid</i> [mg/100 g]
Vit_B6	unreal , — <i>Vitamin B6</i> [mg/100 g]
Folate_Tot	unsigned , — <i>Folate, total</i> [g/100 g]
Folic_acid	unsigned , — <i>Folic acid</i> [g/100 g]
Food_Folate	unsigned , — <i>Food folate</i> [g/100 g]
Folate_DFE	unsigned , — <i>Folate</i> [g dietary folate equivalents/100 g]
Vit_B12	unreal , — <i>Vitamin B12</i> [g/100 g]
Vit_A_IU	unsigned , — <i>Vitamin A</i> [IU/100 g]
Vit_A_RAE	unsigned , — <i>Vitamin A</i> [g retinol activity equivalents/100g]
Retinol	unsigned , — <i>Retinol</i> [g/100 g]
Vit_E	unreal , — <i>Vitamin E (alpha-tocopherol)</i> [mg/100 g]
Vit_K	unreal , — <i>Vitamin K (phylloquinone)</i> [g/100 g]
Alpha_Carot	unreal , — <i>Alpha-carotene</i> [g/100 g]
Beta_Carot	unreal , — <i>Beta-carotene</i> [g/100 g]
Beta_Crypt	unreal ,

5.5. ANNOTATED DATABASE SCHEMA

```

Lycopene          -- Beta-cryptoxanthin [g/100 g]
                  unreal,
Lut_Zea           -- Lycopene [g/100 g]
                  unreal,
                  -- Lutein + zeaxanthin [g/100 g]
FA_SAT            unreal,
                  -- Saturated fatty acid [g/100 g]
FA_Mono           unreal,
                  -- Monounsaturated fatty acids [g/100 g]
FA_Poly           unreal,
                  -- Polyunsaturated fatty acids [g/100 g]
Cholesterol1      unsigned,
                  -- Cholesterol [mg/100 g]
GmWt_1            unreal,
                  -- First household weight for this item
                  -- from the Weight file.
                  -- Weights are given for edible material
                  -- without refuse, that is, the weight
                  -- of an apple without the core or stem,
                  -- or a chicken leg without the bone,
                  -- and so forth.
GmWt_Desc1        VARCHAR,
                  -- Information is provided on household
                  -- measures for food items (for example,
                  -- 1 cup, 1 tablespoon, 1 fruit, 1 leg).
                  -- This is the description of household
                  -- weight number 1.
GmWt_2            unreal,
                  -- Second household weight for this item
                  -- from the Weight file.
GmWt_Desc2        VARCHAR,
                  -- Description of household weight number 2.
Refuse_Pct        unsigned,
                  -- Percentage of refuse.
                  -- Duplicate of food_des.Refuse.
Shrt_Desc         VARCHAR
                  -- Abbreviated description of the food item.
                  -- Duplicate of food_des.Shrt_Desc.
);

-- Just like TABLE abbrev, except that the data here is not
-- from USDA but from our lovely dietary experts.
-- [TODO] Shouldn't we CREATE UNIQUE INDEX?
CREATE TABLE abbrev_menugene () INHERITS (abbrev);
--- Tables }}}

----- usda_am ----- }}}

----- menugene_am ----- {{{

```

```
-- This schema is intended to be the same as the "menugene" schma,
-- except for some syntactic sanitization. I may rename and reorganize
-- DB objects, change types and such, but the structure will be kept.
--
-- [Maintainer: adam]
CREATE SCHEMA menugene_am;
SET search_path TO menugene_am, public;

-- Prerequisites {{{
-----

-- One day these things should land in SCHEMA main.

-- Database entities are identified unambiguously by objid.
-- Whenever you need to reference from a relation to an entity
-- use the objid type. The type doesn't CHECK (VALUE >= 0)
-- because an objid should always either be generated by
-- SEQUENCE objids (which guarantees non-negativeness)
-- or be a reference to another objid, which is unsigned
-- by recursion. Use of the objid type is not just semantic
-- sugar: sequences return BIGINT:s, which you need to be
-- aware of, because BIGINT is wider than INTEGER.
-- (It's worth noting that Postgres generates INTEGER type
-- columns when you use SEQUENCE — looks like a bug.)
CREATE DOMAIN objid AS BIGINT;
CREATE SEQUENCE objids;

-- Creates the necessary unique indices on table $1:
-- $1_index_id for $1.id and $1_index_name_* for $1.name[*].
-- The indices are necessary to guarantee uniqueness
-- of those columns.
CREATE FUNCTION mk_entity_tbl(VARCHAR) RETURNS VOID AS
$$
DECLARE
    tbl      ALIAS FOR $1;
    stmt     VARCHAR;
BEGIN
    -- Create the objid index. We use EXECUTE because
    -- neither SQL not PL/pgSQL allows variable identifiers
    -- at arbitrary places. Look up my ferke project
    -- to gain impression how these things can go wild.
    stmt := 'CREATE UNIQUE INDEX '
            || tbl || ' _index_id '
            || 'ON ' || tbl || '(id);';
    EXECUTE stmt;

    -- Create the developer name index.
    -- (Arrays are subscripted from one.)
```

```

stmt := 'CREATE UNIQUE INDEX '
      || tbl || '_index_name '
      || 'ON ' || tbl || '(LOWER(name[1]));';
EXECUTE stmt;

-- Create the English name index.
stmt := 'CREATE UNIQUE INDEX '
      || tbl || '_index_name_en '
      || 'ON ' || tbl || '(LOWER(name[2]));';
EXECUTE stmt;

-- Create the Hungarian name index.
stmt := 'CREATE UNIQUE INDEX '
      || tbl || '_index_name_hu '
      || 'ON ' || tbl || '(LOWER(name[3]));';
EXECUTE stmt;
END
$$ LANGUAGE PLPGSQL;
-- Prerequisites }}}

-- Tables {{{
-----

-- Entities {{{
-- Entities are entities you know from database theory.
-- This table is intended to be the base of the specific
-- entity tables INHERITS:ing from this one. The pro
-- of this approach is that you needn't bother with
-- requisite fields like entity ID or name, and you can
-- attach arbitrary notes to entities. Besides, whenever
-- you have a random objid it becomes easier to determine
-- what type of entity does it refer to.
--
-- To create a new entity type, make it INHERTS from this
-- one, then create the necessary indices by calling
-- mk_entity_tbl('new_table_name').
--
-- Nothing is ever INSERT:ed into this table directly:
-- all information is in the descendant entity tables.
-- That's why we omit the unique constraints from the
-- definition, because those imply the creation of
-- UNIQUE INDEX:es, which we wouldn't use anyway.
CREATE TABLE entities (
    id                objid                NOT NULL
                                -- PRIMARY KEY
                                DEFAULT NEXTVAL('objids'),
                                -- The unambiguous ID of the entity.
    name              VARCHAR[],          -- UNIQUE
                                -- Textual name (human-readable or not)

```



```

    -- of the entity. Each element of
    -- the array is designated for a
    -- language. ATM:
    -- -- name[1]: temporary name
    -- -- name[2]: English
    -- -- name[3]: Hungarian
    -- mk_entity_tbl() will see to the
    -- uniqueness of the names per language
    -- per entity type.
notes      TEXT
    -- Arbitrary unstructured notes
    -- concerning the entity for the
    -- developer's pleasure.
);

-- Registry of measurement units used to express quantities.
-- (The name of this table is somewhat misleading.)
CREATE TABLE measures (
    in_kg          unreal          NOT NULL
    -- One unit of this is how much in kilogram.
) INHERITS (entities);
SELECT mk_entity_tbl('measures');

-- Registry of existing solutions.
--
-- Levels of genomes:
-- L1 weekly menu,
-- L2 daily menu,
-- L3 meal (breakfast, lunch, dinner, ...).
--
-- Levels of non-genomes:
-- L4 recipe (pizza),
-- L5 recipe component (white bread),
-- L6 nutrition component (or whatever we call it now) (vitamin A).
CREATE TABLE sol (
    is_genome      BOOLEAN          NOT NULL,
    -- Breed in a GA?
    measureid      objid
    REFERENCES measures(id)
    -- Makes no sense above L5, so it can be NULL.
) INHERITS (entities);
SELECT mk_entity_tbl('sol');

-- Registry of attributes like "menu for Monday", "dessert of a meal",
-- "vitamin A content of a recipe component" or "a component of a recipe".
CREATE TABLE attr () INHERITS (entities);
SELECT mk_entity_tbl('attr');

-- Registry of all existing solution sets, which group recipes
```

5.5. ANNOTATED DATABASE SCHEMA

```
-- by arbitrary criteria. Sets may have subsets (also included
-- in this table), constituting a tree-like hierarchy. Relations
-- between sets are recorded in TABLE c_solset_solset.
CREATE TABLE solset () INHERITS (entities);
SELECT mk_entity_tbl('solset');

-- Registry of constraints. Used to be TABLE constraintsol.
-- Constraints play role when you compute the fitness of a
-- solution genome. It's not stored in the database whose
-- (referred to as 'parent' below) fitness will use a given
-- constraint (or constraint set), because that is chosen
-- by the user in run time.
-- .{min,opt,max}imum have been made unreal because quantities
-- in USDA are unreal too.
CREATE TABLE cint (
    solid                objid                NOT NULL
                        REFERENCES sol(id),
                        -- What does this cint constraint?
                        -- Makes no sense to REFERENCE to
                        -- a sol.is_genome (L1..3), alas
                        -- we can't CHECK for it.
    minimum              unreal                NOT NULL,
                        -- The smallest quantity of .solid
                        -- in the parent we'd like to see.
                        -- Used to be .minnum.
    optimum              unreal                NOT NULL,
                        -- The user will want that much
                        -- of .solid in the parent.
                        -- Used to be .optnum.
    maximum              unreal                NOT NULL
                        -- The smallest quantity of .solid
                        -- in the parent we tolerate.
                        -- Used to be .maxnum.
) INHERITS (entities);
SELECT mk_entity_tbl('cint');

-- Registry of existing constraint sets.
-- Used to be TABLE constraintset.
CREATE TABLE cset () INHERITS (entities);
SELECT mk_entity_tbl('cset');

-- Registry of constraint-dividing rules.
CREATE TABLE div () INHERITS (entities);
SELECT mk_entity_tbl('div');
--- }}}

-- Relations {{{
-- Tells what attributes a solution has.
CREATE TABLE c_sol_attr (
```

5.5. ANNOTATED DATABASE SCHEMA

```
solid          objid          NOT NULL
               REFERENCES sol(id),
attrid         objid          NOT NULL
               REFERENCES attr(id),
quantity       unreal         NOT NULL,
               -- E.g. how much white bread does a pizza contain?
               -- Used to be .defnum.
min            unreal,
               -- Unreasoned and unused by now.
               -- Used to be .minnum.
max            unreal,
               -- Unreasoned and unused by now.
               -- Used to be .maxnum.
divid          objid
               REFERENCES div(id),
               -- Tells how to distribute the constraints applied
               -- to this solution among its different attributes.
UNIQUE (solid, attrid)
);

-- For the quick enumeration of the attributes of a particular solution.
CREATE INDEX sol_attr_index ON c_sol_attr (solid);

-- Tells what values an attribute can present.
-- In case of ATOM attributes there will be only one,
-- pointing to a recipe. For POOL:s there can be many,
-- pointing to recipes as well. Finally a GA attribute
-- has only one solid, which .is_genome.
-- [TODO] soltype was not part of the menugene schmea,
-- don't know what's happened to that.
CREATE TABLE c_attr_sol (
    attrid      objid          NOT NULL
               REFERENCES attr(id),
    solid       objid          NOT NULL
               REFERENCES sol(id),
    UNIQUE (attrid, solid)
);

-- For the quick enumeration of the solutions of a particular attribute.
CREATE INDEX attr_sol_index ON c_attr_sol (attrid);

-- Keeps the parent-child relationships between solsets,
-- i.e. solset .subsetid is a child of solset .setid.
-- Used to be TABLE solsubset.
--
-- Too bad PostgreSQL doesn't support recursive queries
-- (MS SQL does, for years) so we must express CHECK
-- against circular dependencies in human terms:
-- if solset A IS a child of solset B
```

5.5. ANNOTATED DATABASE SCHEMA

```
--      then solset B IS NOT a child of solset A.
CREATE TABLE c_solset_solset (
    setid          objid          NOT NULL
                    REFERENCES solset(id),
                    -- Parent solset.
    subsetid       objid          NOT NULL
                    REFERENCES solset(id),
                    -- Child solset.
    UNIQUE (setid, subsetid)
);

-- For the quick enumeration of the children of a solset.
CREATE INDEX solset_subset_index ON c_solset_solset (setid);

-- Tells what solutions (recipes mostly) constitute a solset.
CREATE TABLE c_solset_sol (
    setid          objid          NOT NULL
                    REFERENCES solset(id),
    solid          objid          NOT NULL
                    REFERENCES sol(id),
    UNIQUE (setid, solid)
);

-- For the quick enumeration of the members a solset.
CREATE INDEX solset_sol_index ON c_solset_sol (setid);

-- Suppose you've got a genome solution, some constraints applied to it.
-- Each of its relations with its attributes are assigned a dividing rule
-- (c_sol_attr.divid) that tells how to distribute its constraints among
-- the solutions of the attributes. This rule is described in this table.
--
-- [TODO] What is .fitnesstype doing here?
-- [TODO] The purpose of .divtype == 0 is not clear.
-- [TODO] The meaning of .divtype == 2 is not clear at all.
CREATE TABLE c_div_sol (
    divid          objid          NOT NULL
                    REFERENCES div(id),
    divtype        INTEGER        NOT NULL
                    CHECK (divtype BETWEEN 0 AND 2),
                    -- Tells how to derive the constraints of the
                    -- child solutions.
                    -- 0: use .solid, .numerator, .denominator
                    -- [TODO just guessing]
                    -- 1: use .solid, .multiply, .tolerance
                    -- 2: use .cint or .cset
                    -- [TODO just guessing]
    fitnesstype    VARCHAR
                    CHECK (fitnesstype IN ('tdk', 'pb')),
                    -- [TODO] Should be in TABLE sol instead.
```

5.5. ANNOTATED DATABASE SCHEMA

```

-- For .divtype == 0 or 1
solid          objid
               REFERENCES sol(id),

-- For .divtype == 0
numerator      unsigned,          -- Unused by daemon.
denominator    unsigned,          -- Unused by daemon.

-- For .divtype == 1
multiply       unreal,
               -- Multiply the constraint applied to .solid
               -- by this.
               -- Used to be .percentage. Renamed because
               -- it never was interpreted as a percentage.
tolerance       unreal,
               -- Multiply the distance of .{min,max}imum
               -- from opimum by this. (Adjust the tolerated
               -- range of the constraint applied to .solid).
               -- Used to be .hardness. Renamed because
               -- daemon called it 'tolerance'.

-- For .divtype == 2
cint           objid              -- Unused by daemon.
               REFERENCES cint(id),
cset           objid              -- Unused by daemon.
               REFERENCES cset(id),

CHECK (CASE divtype -- [TODO] Find a simpler expression.
      WHEN 0 THEN
          (solid      IS NOT NULL
           AND multiply IS NOT NULL)
      WHEN 1 THEN -- [TODO] Just guessing.
          (solid      IS NOT NULL
           AND numerator IS NOT NULL
           AND denominator IS NOT NULL)
      WHEN 2 THEN -- [TODO] Just guessing.
          (cint       IS NOT NULL
           OR cset     IS NOT NULL)
      ELSE FALSE
      END),
UNIQUE (divid, solid)
);
-- Relations }}}
-- Tables }}}

----- menugene_am ----- }}}

```