Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

Abril 2020

Pedro Costa (A85700)

Miguel Carvalho (A84518)

Maria João Moreira (A89540)

Conteúdo

1 Introdução **2**

2 Módulos e funcionalidades correspondentes **2**
2.1 Componentes principais do SGV - Visão superficial 2
2.1.1 Catálogo - A estrutura de dados mais eficiente para a nossa aplicação 3
2.1.2 Catálogo de Produtos/Clientes 4
2.1.3 Faturação 4
2.1.4 Filial 4
2.2 Outros módulos desenvolvidos por necessidade/utilidade 5
2.2.1 Parser 5
2.2.2 Config 5
2.2.3 Navegador 5
2.2.4 Pair 5
2.2.5 PQueue 5
2.2.6 Sale 5
2.2.7 Stats 6
2.2.8 Encapsulamento 6

3 Arquitetura da aplicação **7**

4 Otimizações e profiling **8**

5 Utilização do programa e melhorias a fazer **10**

6 Testes realizados **10**

7 Conclusão **11**

1 Introdução

Tal como é dito no enunciado deste projeto, o desenvolvimento desta aplicação tem por objetivo consolidar, com ênfase na parte prática, os conhecimentos adquiridos noutras UCs (nomeadamente *Algoritmos e Complexidade*) e, ainda, introduzir **novos princípios de programação** aplicáveis à programação em média-larga escala.

Tal como em vários projetos de média-larga escala, no desenvolvimento desta aplicação lidamos com grandes volumes de dados e foi imperativo ter em consideração o uso de estruturas adequadas para armazenar os ditos dados e ser capazes de os consultar de forma eficiente.

2 Módulos e funcionalidades correspondentes

2.1 Componentes principais do SGV - Visão superficial

O módulo principal da nossa aplicação, o **SGV** é composto pelas seguintes componentes, tal como foi pedido.

- **Catálogo de Produtos**: Módulo de dados onde se guarda todos os códigos de produtos válidos lidos.
- **Catálogo de Clientes**: Igual ao Catálogo de Produtos mas guarda códigos de clientes.
- **Faturação**: Módulo de dados que relaciona produtos com informação relativa às suas vendas (quantidades vendidas e lucros).
- **Filial**: Módulo de dados que relaciona clientes com os produtos que compram para **uma** dada filial (saber quanto comprou e quanto gastou em todos os produtos que obteve).

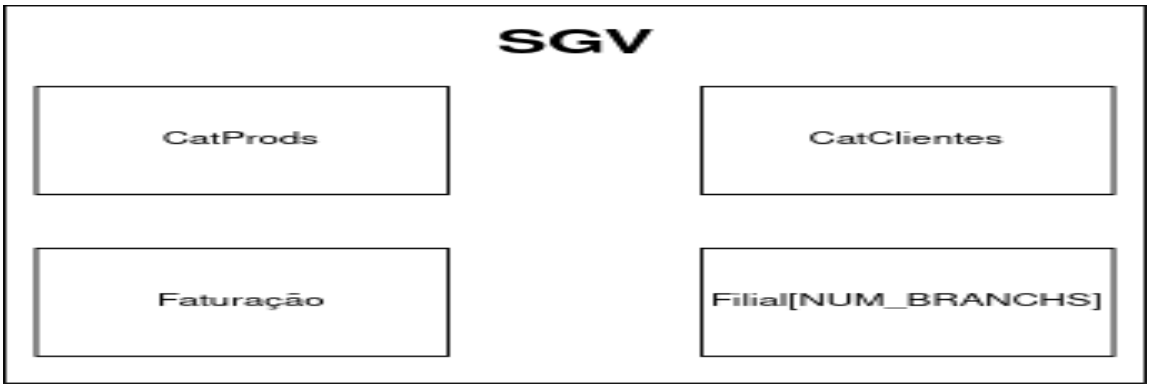

Figura 1: Componentes do SGV

2.1.1 Catálogo - A estrutura de dados mais eficiente para a nossa aplicação

Ainda que não seja uma das estruturas principais do SGV, o módulo de dados **Catalog** é, sem sombra de dúvidas, o ponto central do nosso projeto. Todas as componentes do SGV foram definidas à custa deste módulo uma vez que permite acessos extremamente rápidos aos códigos que contém.

Este módulo é composto por um array de GTree* em que cada *tree* corresponde a todas as chaves começadas por um conjunto de letras (*ie.* "A", "ZD", "FRW", etc...).

Esta estrutura assemelha-se muito a um dicionário, catalogando as chaves que contém de acordo com a sua nomenclatura. O esquema abaixo explica melhor como está desenhado este módulo.

Nota: Todos os valores inseridos nas trees tem chave correspondente à parte numérica do código introduzido.

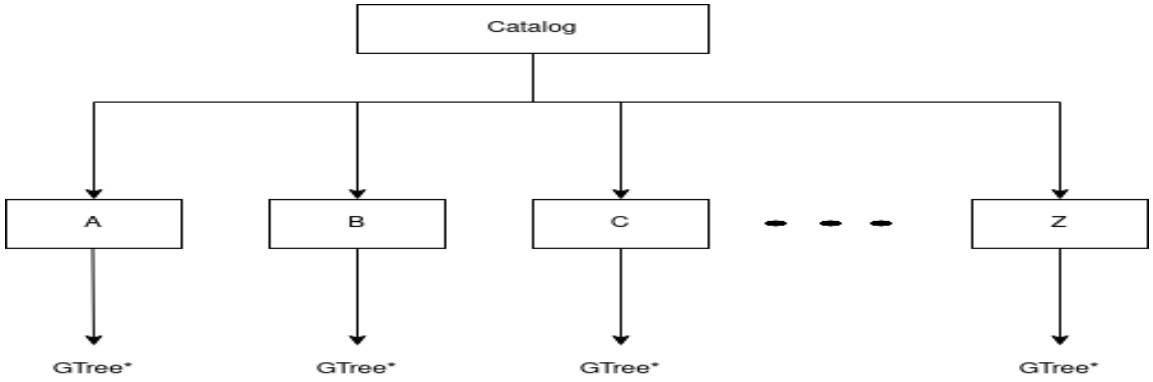

Figura 2: Catalog

Esta estrutura tem uma complexidade de $O(1+\log_2 n)$, em que n corresponde ao maior tamanho possível de cada tree. A tabela seguinte mostra uma análise do valor de n para o caso dos clientes e produtos.

Tipo	LL	UL	n	Nº max acessos
Produto	1000	9000	8000	$1+\log_2 8000 = 14$
Cliente	1000	5000	4000	$1+\log_2 4000 = 13$

Legenda: LL = Lower limit; UL = Upper limit

2.1.2 Catálogo de Produtos/Clientes

Ambos estes módulos são compostos unicamente por um catálogo cujas chaves correspondem à parte númerica código do cliente ou produto em causa. Em ambas as situações, o campo value corresponde à string completa do código.

2.1.3 Faturação

A faturação é, como indicado anteriormente, também composta unicamente por um Catalog. Nesse, para cada produto, guarda-se a *struct* indicada abaixo:

```
struct productInfo{
    /* Código do produto */
    char* product;
    /* Vendas distribuídas por tipo, filial e mês */
    int sales[2][NUM_BRANCHES][NUM_MESES];
    /* Lucros distribuídos por tipo, filial e mês */
    float profit[2][NUM_BRANCHES][NUM_MESES];
    /* Total de vendas por tipo (útil mas desnecessário) */
    int total_sales[2];
    /* Total de lucro por tipo (útil mas desnecessário) */
    int total_profit[2];
};
```

2.1.4 Filial

Este módulo é também implementado com base num catálogo. Para cada cliente guarda-se uma **GHashTable*** que armazena a seguinte *struct*:

```
struct customerbought{
    /* Código do produto comprado */
    char* product;
    /* Número de compras por tipo ao longo dos meses */
    int n_compras[2][NUM_MESES];
    /* Lucro por tipo ao longo dos meses */
    float total_gasto[2][NUM_MESES];
};
```

Nota: Apenas este módulo tem uma complexidade ligeiramente mais elevada que os anteriores uma vez que para se obter um valor é necessário fazer uma procura no catálogo e, de seguida, uma procura na *tree* que se obtém.

2.2 Outros módulos desenvolvidos por necessidade/utilidade

2.2.1 Parser

Este módulo tem unicamente o objetivo de fazer parse. Isto implica as seguintes duas funcionalidades:

- **Validação**: Disponibiliza funções capazes de validar códigos de produtos e de clientes. Útil no parse efetuado no controller e dentro do próprio carregamento.
- **Carregamento de dados**: Disponibiliza as funções **loadCatalog** e **loadVendas** para carregar catálogos e vendas, respetivamente, a partir de caminhos para os ficheiros a ler.

2.2.2 Config

Este módulo contém várias funções úteis em mais do que um módulo desenvolvido. Assim evitamos duplicação desnecessária de código.

2.2.3 Navegador

Módulo de dados utilizado para navegar grandes quantidades de informação. A sua implementação extremamente simples baseia-se inteiramente em ter um inteiro que marca o sítio de onde se deve a começar a imprimir duma dada lista de strings e em saber que quantidade é que se deve imprimir de cada vez. Consoante o utilizador pede basta subtrair/somar ao ponto de partida o tamanho que imprimos de cada vez.

2.2.4 Pair

Tal como o nome indica, esta API define apenas um par. É uma estrutura extremamente útil uma vez que várias vezes nos encontramos numa situação em que queríamos retornar mais do que um valor (com tipos diferentes) sem estar a devolver um *void** ou aumentar desnecessariamente o número de argumentos.

2.2.5 PQueue

Este módulo foi desenvolvido a partir do ficheiro que se encontra na pasta *libs*. Utilizámo-lo nas queries 10, 11 e 12 pois queríamos devolver algo com base numa certa prioridade. Pareceu-nos apenas natural utilizar uma *priority queue* para tal efeito. Fizemos algumas trocas ao original com base em pormenores que nos facilitariam a vida tais como:

- Organizar as entradas em key value. Permite-nos utilizar funções de comparação mais simples que não precisam de conhecer o value na íntegra.
- Permitir o fornecimento de funções de free das keys e dos values para se limpar automaticamente toda a memória alocada ao fazer destroy e ao fazer dequeue.
- Queríamos ainda ter implementado a possibilidade de fazer resize caso se ultrapasse a capacidade máxima. Não foi implementada esta feature.

2.2.6 Sale

Este módulo define apenas o que é uma venda disponibilizando na sua API funções de inicialização e *gets* para se obter os respetivos valores armazenados.

2.2.7 Stats

Ainda que este módulo seja uma componente do SGV, não é essencial para o seu funcionamento pelo que se trata dum móduloe extra. O *Stats* surgiu para satisfazer uma necessidade extremamente simples. Cronometrar qualquer coisa que quisessemos. O código para cronometrar uma função ainda implica algumas linhas pelo que iríamos poluir muito a nossa base de código para fazer *profiling* em condições. Assim, definimos uma maneira simples para efetuar esta tarefa. Basta adicionar à função *init* os nomes do que queremos cronometar e, quando desejarmos, chamar as seguintes funções.

```
/* Starts the timer for the given key */
void start_timing(Stats s, const char* key);

/* Agreggates the time into a given key */
void agreggate_timing(Stats s, const char* key);

/* Finishes timing and writes the result to the file associated with a key */
void finish_timing(Stats s, const char* key);
```

A função *agreggate_timing* permite cronometrar funções pelas quais passamos mais duma vez, agregando os novos valores a cada utilização.

2.2.8 Encapsulamento

Tal como pedido, garantimos que todos os nossos módulos (à exceção do Stats) têm implementações privadas que nunca permitem que qualquer utilizador consiga alterar o SGV duma maneira não desejável. Para esse efeito não abordamos pela abordagem OO mas sim por criar API's mais extensas que permitem ao utilizador interessado (neste caso, o próprio grupo) obter qualquer informação necessária através do SGV.

3 Arquitetura da aplicação

Foi sugerido que adotássemos uma arquitetura MVC para unir as diversas componentes da nossa aplicação final e, sendo assim, foi exatamente isso que fizemos. No entanto, devido ao que foi explicado na secção *Encapsulamento* temos um pormenor diferente do que seria uma implementação típica de MVC. No nosso caso, é o próprio SGV que manipula os dados internamente para responder às várias queries. O esquema abaixo demonstra bem as interações no nosso projeto.

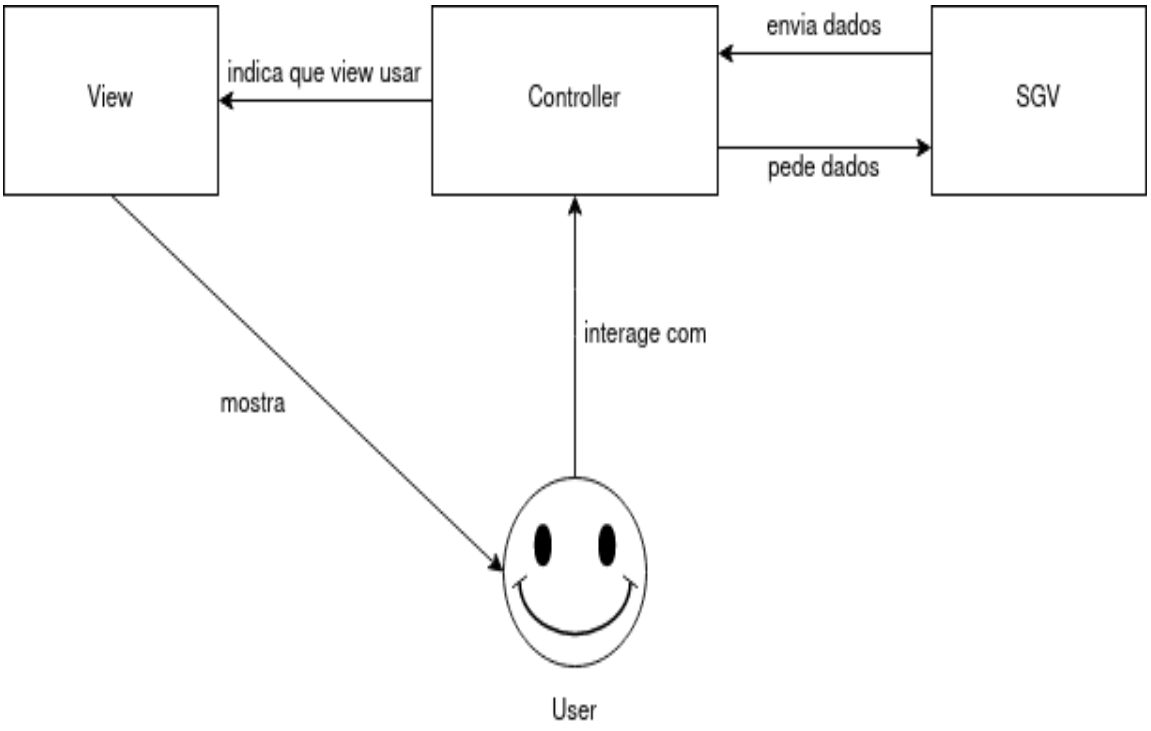

Figura 3: **Design pattern - MVC**

4 Otimizações e profiling

Tal como referido anteriormente, todo o nosso projeto assenta no módulo de dados **Catálogo**. Assim, é apenas natural que a maneira mais eficaz de otimizar todo o programa passe por otimizar esse módulo. Assim, oferecemos uma implementação extra do projeto que utiliza um catálogo extremamente mais eficiente mas que consome muita mais memória. Esta implementação encontra-se numa pasta que apelidamos por **Performance_Mode** e é um fork do projeto original numa fase em que já se conseguia dar resposta a todas as queries.

A figura abaixo ajuda a compreender a diferença para o **Catálogo** do projeto principal enviado.

Nota: Esta estrutura tem, literalmente, acessos constantes a cada valor (O(1)).

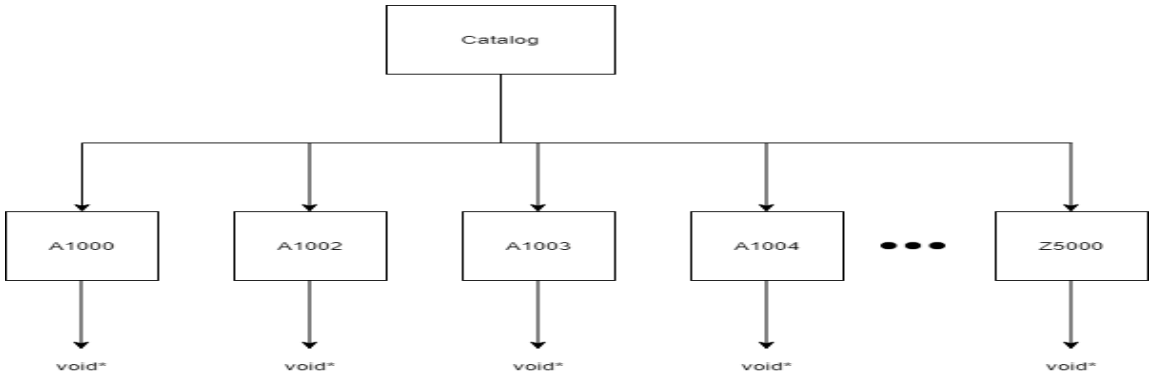

Figura 4: Optimized catalog

Vamos agora comparar exaustivamente todos os tempos medidos (**Nota**: todos os tempos apresentados são *elapsed times*).

Legenda: TP = Trabalho prático principal; PM = Performance Mode; XM = Ficheiros Vendas_XM.txt

Parsing	TP_1M	PM_1M	TP_3M	PM_3M	TP_5M	PM_5M
billProduct	0.417208	0.263402	1.518986	0.874880	2.396639	1.532954
cBought	0.858921	0.626829	3.200031	2.151543	4.974849	3.681399
loadCC	0.042927	0.012049	0.024715	0.012149	0.023263	0.012020
loadCP	0.137195	0.068430	1.54743	0.068709	0.134901	0.068051
loadVendas	3.577178	2.003663	12.628885	6.365291	19.606948	10.791212
catExists	1.384713	0.193839	4.765775	0.570989	7.393718	0.985675
words	0.297074	297820	0.993529	0.870838	1.521031	1.469308
totalLoad	3.75	1.95	11.77	5.63	17.92	9.46

O que se vê de interessante na tabela acima? Principalmente 3 coisas.

- Devido a cronometrar funções que são chamadas uma vez por linha das vendas nota-se uma inflação de cerca de 10% no tempo registado pelo *loadVendas* em comparação com o *totalLoad* quando estes, em teoria, seriam equivalentes.

- O grande peso do carregamento centra-se em verificar que os códigos de produto/cliente existem nos catálogos. Em média, com a otimização feita no **Performance_Mode** regista-se um aumento entre 7-10 vezes no tempo dessa ação.
- A otimização feita no **Performance_Mode** regista praticamente metade do tempo total de carregamento em relação ao projeto principal. Uma diferença abismal sem dúvida.

Queries	TP_1M	PM_1M	TP_3M	PM_3M	TP_5M	PM_5M
query2	0.002735	0.001739	0.002209	0.001665	0.002723	0.001681
query3	0.000023	0.000001	0.000005	0.000001	0.000012	0.000001
query4	0.202662	0.193588	0.198692	0.199581	0.256908	0.203211
query5	0.006563	0.004580	0.015275	0.004275	0.007031	0.004717
query6	0.065913	0.043696	0.076181	0.039667	0.066692	0.043802
query7	0.000016	0.000008	0.000038	0.000021	0.000068	0.000034
query8	0.345657	0.132062	0.108621	0.131767	0.385788	0.126734
query9	0.006245	0.003522	0.004330	0.003924	0.007911	0.004902
query10	0.00003	0.000013	0.000113	0.000026	0.000074	0.000035
query11	0.480123	0.455098	1.279723	1.170635	1.914258	1.914663
query12	0.000155	0.000046	0.000155	0.000083	0.00056	0.000140

O que se vê de interessante na tabela acima? E qual a diferença principal que se nota em comparação com a tabela de parsing?

- Nesta tabela acontece uma de duas coisas quando comparamos os tempos entre os dois modos diferentes. Ou são iguais, ou o **Performance_Mode** é praticamente 2 vezes mais rápido. O primeiro caso deve-se ao facto de a maior parte das queries serem travessias, pelo que não faz diferença o tempo de acesso a cada campo. O segundo caso deve-se ao facto de o acesso à memória ser constante no **Performance_Mode**.
- A principal com a tabela de parsing é exatamente o que foi dito acima. Os tempos aqui são extremamente parecidos uma vez que não estamos a lidar tanto com acessos à memória, tirando menos proveito da otimização feita.

5 Utilização do programa e melhorias a fazer

A aplicação que desenvolvemos oferece também ao utilizador três flags extremamente úteis para uso continuado.

- —**timing-mode**: Tal como o nome indica, esta flag ativa o timing mode, gravando nos ficheiros correspondentes os tempos obtidos ao longo do programa. (**Nota**: Tem obrigatoriamente de ser a primeira flag.)
- —**save-queries**: Esta flag permite que todas as queries, além de serem demonstradas na UI ao utilizador, sejam ainda gravadas em ficheiros de texto. Extremamente útil para comparação de resultados.
- —**default-load**: Esta, a nossa favorita, permite pré-carregar o programa sem ter de especificar um a um os ficheiros a serem utilizados.
- —**config**=**configfile**: Uma opção extremamente prática que permite fornecer alguns dados ao programa através de um ficheiro com configurações. Caso não seja incluída o programa assume um conjunto de predefinições.

A nível de melhorias do nosso projeto achamos que não há muito a fazer. Possivelmente melhorar a interface gráfica e adicionar features ao navegador seriam um bom ponto de partida (ex: 2n para andar 2 páginas).

6 Testes realizados

Ao longo do desenvolvimento do projeto testamos unitariamente cada módulo separadamente de maneira a garantir que sempre que adicionamos mais código à nossa base, caso houvesse algum erro, este encontrar-se-ia no código adicionado mais recentemente. Assim conseguimos potencialmente nunca ter de fazer uma grande regressão. Numa fase mais avançada, cruzamos os nossos dados com os oferecidos no ficheiro **"Resultados para comparação"**.

Ao longo do seu desenvolvimento, todo o código desenvolvido foi testado na ferramenta **valgrind** para garantir que não existem quaisquer leaks de memória ou acessos ilegais à mesma.

```
6 - Numero de clientes que nao realizaram compras e produtos nao comprados
7 - Tabela com numero total de produtos comprados ao longo dos meses e nas distintas filiais por um cliente
8 - Total de vendas e total faturado num intervalo de meses
9 - Codigo e numero de clientes que compraram um produto numa filial
10 - Lista de produtos que um cliente mais comprou num mes por ordem decrescente
11 - Lista dos produtos mais vendidos no ano
12 - Produtos em que o cliente mais gastou no ano
13 - Apresentaçao da leitura de ficheiros (Produtos, Clientes e Vendas)
14 - Informação acerca de um dado produto
0
==225415==
==225415== HEAP SUMMARY:
==225415==     in use at exit: 19,084 bytes in 7 blocks
==225415==   total heap usage: 5,267,498 allocs, 5,267,491 frees, 167,618,749 bytes allocated
==225415==
==225415== LEAK SUMMARY:
==225415==    definitely lost: 0 bytes in 0 blocks
==225415==    indirectly lost: 0 bytes in 0 blocks
==225415==      possibly lost: 0 bytes in 0 blocks
==225415==    still reachable: 19,084 bytes in 7 blocks
==225415==         suppressed: 0 bytes in 0 blocks
==225415== Reachable blocks (those to which a pointer was found) are not shown.
==225415== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==225415==
==225415== For lists of detected and suppressed errors, rerun with: -s
==225415== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
➜  src git:(master) ✗
```

Figura 5: Output do valgrind para uma execução do nosso programa

7 Conclusão

Este projeto fez-nos principalmente questionar aquilo que sabemos sobre arquiteturas MVC e encapsulamento. Ao tomar a abordagem diferente que tomamos, aprendemos e questionamo-nos imenso acerca da natureza OO e o que motiva e torna tão simples utilizar esse paradigma para aplicações orientados ao utilizador em média-larga escala.

Foi também uma excelente oportunidade para melhorar os nossos conhecimentos sobre a linguagem de programação **C** e perceber que **Java** não se trata de nada mais que C com um *garbage collector* e com alguma magia escondida. Achamos também interessante a possibilidade de alcançar features maioritariamente utilizadas no paradigma funcional como funções de ordem superior (a *GLib* forçou-nos parte disso) para nos facilitar a vida no desenvolvimento de coisas monótonas e repetitivas, como é o caso do redirecionamento efetuado pelo menu principal.

A nível de falhas do nosso projeto achamos, sem dúvida, que o que nos podia trazer problemas a longo prazo caso o projeto continuasse a escalar é a nossa escolha de arquitetura. Obriga a codificar todas as ações de cada módulo no do SGV pelo que acabamos por ter alguma duplicação desnecessária. Note-se que a arquitetura é efetivamente funcional mas podia estar melhor. Parte da razão de não termos tentado trocar passa-se pelo facto de queremos aproveitar esta oportunidade para experimentar coisas diferentes e poder fazer algo muito sábio, aprender com os erros.

Em suma, este projeto permitiu-nos uma compreensão muito mais profunda de como os diferentes paradigmas se aproximam dos problemas e, além disso, permitiu-nos ver que, com algum esforço adicional, é possível aproximarmo-nos das vantagens de qualquer paradigma com o que quer que seja que estamos a trabalhar.