Process Mix of Items - Mathworks C++ Assignment

Vaidehi Venkatesan

27 March 2013

Abstract

process_mix_of_items code contains a software challenge. The challenge is to apply C++ object oriented software development skills.The zip file process_mix_of_items.zip contains source code for a program that processes a mixture of items. The program is written using C style. This write up document discusses the possible object oriented design strategies considered and implemented for this challenge alongside giving a comparison of the solutions suggested.

1 Overall Idea

This document briefs the four methods / design strategies implemented to apply object oriented design strategies in the given challenge.

1. array data that holds objects of type void*
2. data is filled with objects of three kinds - struct phoneExtension, houseNumber, age.
3. each struct has a member variable - kind apart from struct specific member variables like area code, seven digit number etc.
4. data is printed out by a function called show_data. This function uses reinterpret_cast to cast the void* object to the object kind. Switching on the object kind, the object is once againt reinterpret_cast to appropriate struct type to print information of the member variables.

Underpinning requirement of the code is to have a heterogenous container which can hold objects of different data types namely struct phoneExtension, struct age, struct houseNumber.

A good object oriented re-design of this code, should also have the following principles as foundation.

1. Open-closed principle - entities should be open for extension and closed for modification. In this case, the heterogenous container should be open for accomodating new datatypes but closed in terms of its functionalities

2. Other principles defined in SOLID - Single responsibility, Liskov substitution, Interface segregation, Dependency Inversion should also be kept in mind while coming up with a good design.

3. In order to design with SOLID principles, appropriate design patterns must be employed which can simulateneously achieve the goal to designing a heterogenous C++ container.

2 Solution strategies

The four different ways of coming up with object oriented design for this problem can be seen a incremental approaches from one to the next. A brief description of the four approaches is given below:

1. unionApproach - This approach succeeds in mitigating the use of void* and reinterpret_cast to represent heterogenous data. It uses a vector of union objects containing the information given by the three structs in the original design. The method that prints the information for every union object is overloaded for every given type.

2. itemWrapperApproach - This approach makes use of the fact that all input structs in design have unsigned int or unsigned long data. It defines Item as an object which has type and vector of unsigned longs as member variables. Item has other member functions for printing the relevant data type. It uses overloaded constructors to create the right kind of objects. Thus, encapsulation and object oriented design is handled in this design.

3. inheritenceApproach - This approach uses inheritence and polymorphism to represent given data. All items derive from an abstract base class. The container of items contain pointer to base class. Using virtual member functions, appropriate data is shown.

4. chameleonApproach - This approach uses template member functions which internally create and maintain a static map of <Item*, type> object. The class is not templated while the member functions which set and get value are templated. Hence ,a vector<Item *> is maintained where each Item* can point to object of any data type. This leverages the maximum used of object oriented design in creating a heterogenous container.

3 Class Diagrams

The following class diagrams are drawn to the best approximation using http://www.creately.com

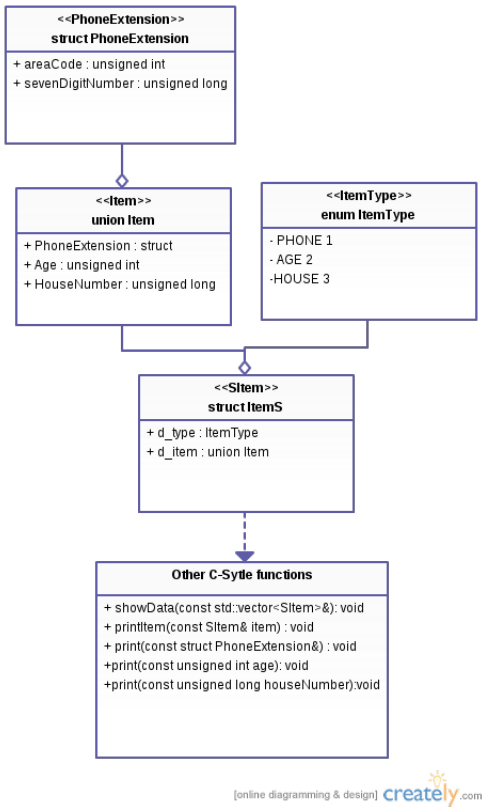

Figure 1: UnionApproach Class Diagram

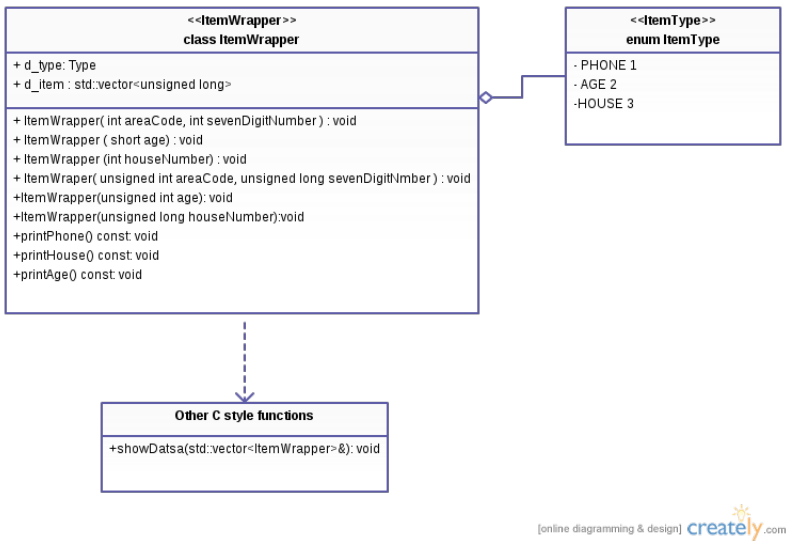

Figure 2: ItemWrapper Class Diagram

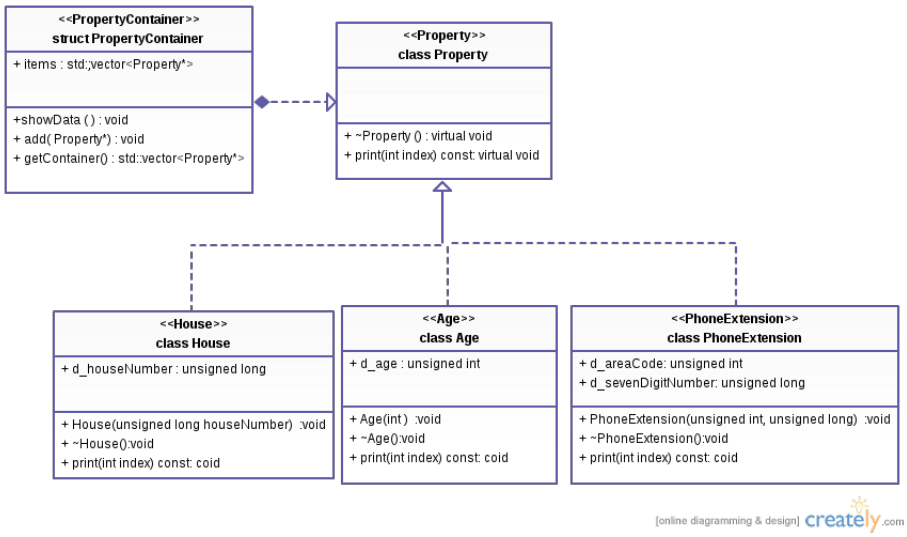

Figure 3: InheritenceApproach Class Diagram

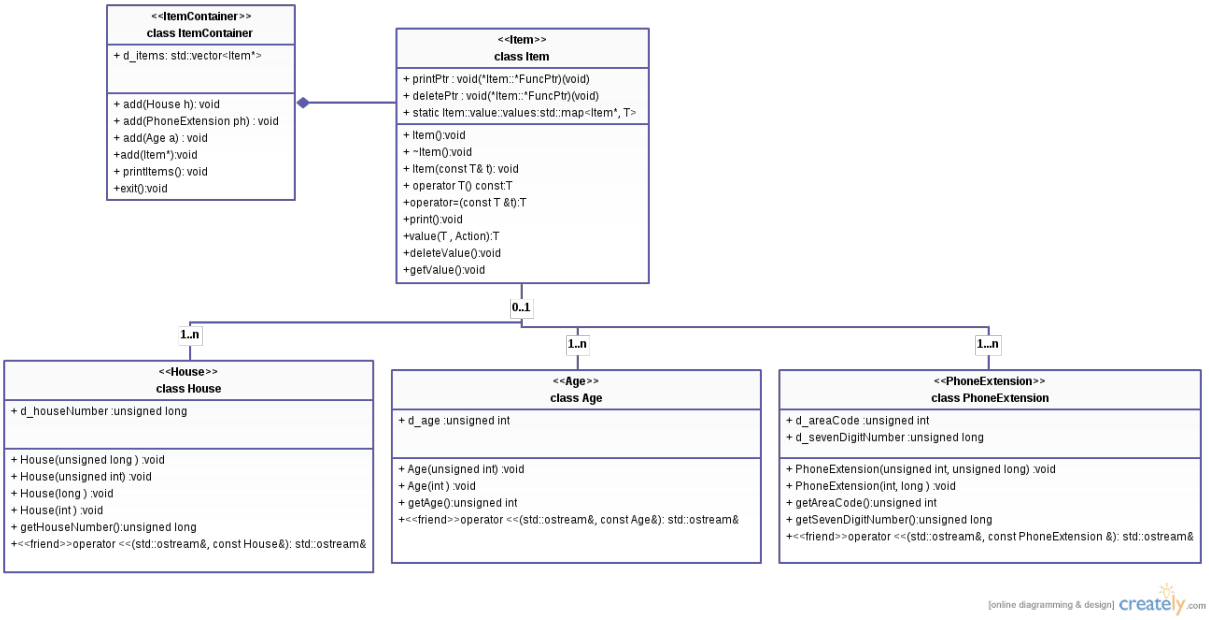

Figure 4: Chameleon Approach Class Diagram

4 Comparison of Approaches

All the approaches attempt to use a generic datastructure to store the heterogenous types with using C++ std::cast or void* objects.

5 Design Patterns

1. Union approach - None
2. ItemWrapper - Adapter
3. Inheritence - Factory Method, Singleton,
4. Chameleon - Singleton, Template Method, Facade

6 References

Approach	Pros	Cons
Union	1. Better Encapsulation 2. No underlying assumption about data	1. Not an OOP approach 2. Open-close principle is not followed 3. Other SOLID principles are also not followed
ItemWrapper	1. OOP approach 2. Encapsulation 3. Single responsibility 4. Interface Segregation	1. There is an underlying assumption about datatypes which makes it non-extensible to other datatypes 2. Open closed principle, Liskov substitution and Dependency Inversion is not followed
Inheritence	1. OOP approach, 2. Encapsulation 3. SOLID Principles are followed	1. Hierarchy of Items is a bad design as there is no underlying real world relationship between the items 2. Bad implementation can lead to memory leaks
Chameleon	1. OOP approach 2. Encapsulation 3. SOLID Principles are followed 4. unambiguous Heterogenous container	1. Use of templates can lead to piling up code in memory 2. A bad implementation can lead to memory leaks.

Table 1: Comparison of four approaches