A Generative Graph Toolkit for C++

Presented
By
Theodore Koopman

Overview

- Objectives
- Differentiating between generic and generative
- Benefits of a generative toolkit
- Planning
- Designing
- Implementing
- Disadvantages
- Conclusion

Objectives

- Explore different ways of representing graphs
- Exploring C++ meta-programming techniques to implement the toolkit
- Discover a naming convention for the graph representations

Differentiating between generic and generative

- Generic
- Generative

Benefits of a generative toolkit

- Compile time checking
- More efficiency
- Relatively easy to maintain
- Elementary components
- Maximum combinability
- Minimum redundancy

Planning

- Use a domain engineering process such as DEMRAL (Domain Engineering Method for Reusable Algorithmic Libraries)
- Define the domain
- Model the domain

Designing

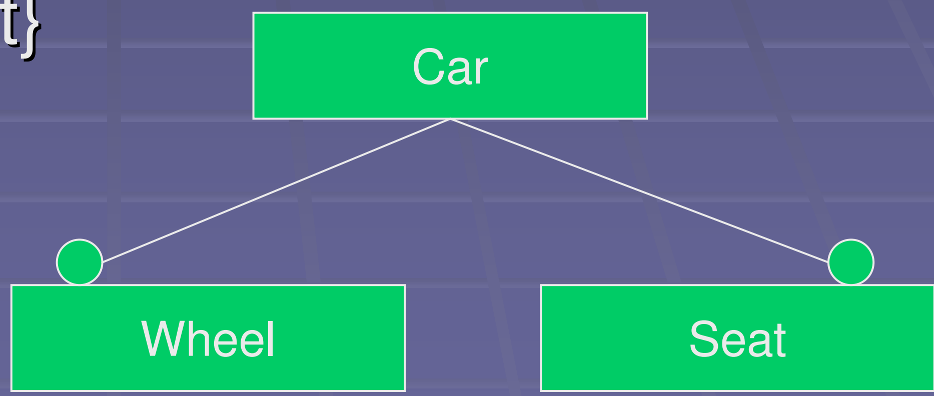
- Specify the toolkit in terms of feature diagrams (feature modeling)
- Feature diagrams
  - Concepts & features
    - Compulsory feature
    - Optional feature
    - Alternatives

Concepts & Features

- Concept
  - Example: Car
- Feature
  - Example: Wheel

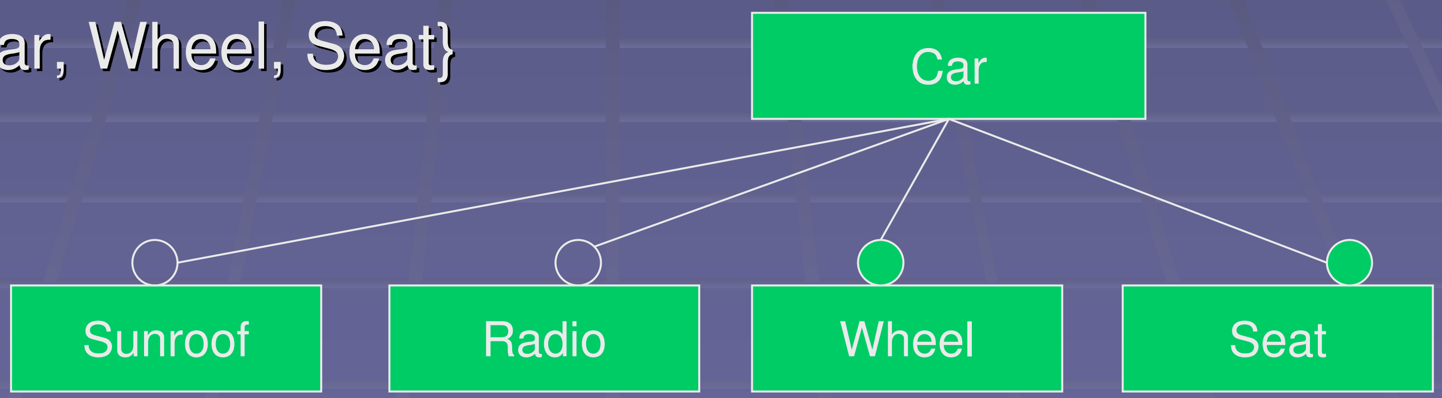
Compulsory Feature

- Denoted by filled circle at the end of connection
- Appears in the set if the parent is in the set
- Example
  - {Car, Wheel, Seat}



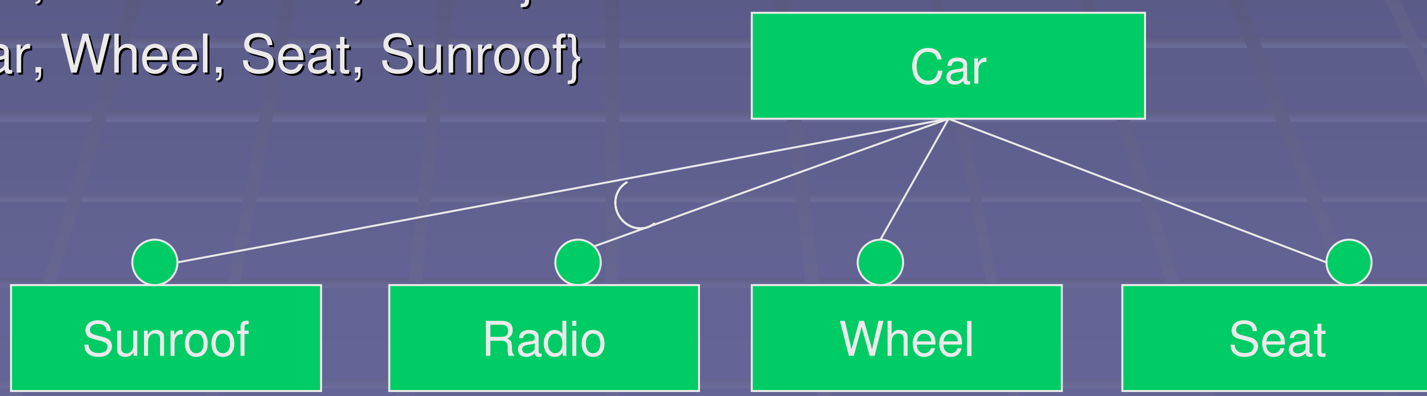
Optional Feature

- Depicted by an unfilled circle
- Can appear in the set if the parent is in the set
- Example
  - {Car, Wheel, Seat, Sunroof}
  - {Car, Wheel, Seat, Radio}
  - {Car, Wheel, Seat}



Alternative Features

- Denoted by an unfilled arc joining two or more features
- Groups features together
- One feature from the group should appear if it is marked as compulsory
- Example
  - {Car, Wheel, Seat, Radio}
  - {Car, Wheel, Seat, Sunroof}



Design In Context

- Concept:
  - Graphs
- Features:
  - Type of container/collection used
  - Type of insertion method used
  - Type of lookup method used
  - Type of isomorphism is used

Graph Design & Modeling

- Graph definition
  - $T = \{N_s, E, N_d\}$
  - $G = \{T_0, T_1, \ldots, T_n\}$
- Types of graphs
  - Linear, binary, mapped or hashed
- Options
  - Isomorphism
    - Left-associated, right-associated, reversed or transposed
  - Insertion policy
    - Insert-at-head, insert-at-tail
  - Lookup policy
    - Move-to-front, migrate-forward, on-found
  - Traversal policy
    - Pre-order, in-order, post-order

Implementing

- Static meta-programming
- Use templates for:
  - Representing meta information
  - Application of meta functions
    - Example of a meta function (The ‘IF’ construct):

```
template<bool Condition, class Then, class Else>
struct IF{ typedef Then Return; };

template<class Then, class Else>
struct IF<false, Then, Else> IF{ typedef Else Return; };
```

Implementation in Context

- Define properties of the graph
- Define the graph class interface
- Individualise algorithms
- Build the graph class generator
- Use the product of the graph class generator

Disadvantages

- Time consuming to build
- Unfriendly error messages generated by compiler
- Constructs used are compiler dependent

Conclusion

- Despite the disadvantages, we are able to produce a graph toolkit that has:
  - A single graph class
  - Different representations
  - Different algorithms
  - With a single interface that is:
    - Relatively easy to maintain and extend
    - Easy to understand and use