Turning Design Patterns into reusable components

Semesterarbeit by Anders Haugeto
Supervised by Karine Amout
Overview

• Introduction / Project goals
• The Pattern Wizard
• The creational Design Patterns
  • Abstract Factory
  • Factory Method
  • Builder
  • Prototype
  • Singleton
• Conclusion
• Evaluation / Questions
Project goal

• Study the Creational Design Patterns
• Evaluate their applicability as reusable components in Eiffel
• Classify each Design Pattern into two groups
  • Applicable as reusable component
  • Not applicable as reusable component
Project goal (2)

- For each Design Pattern applicable as reusable component in Eiffel:
  - Implement it
  - Write usage-explanation and example
- For each Design Pattern not applicable as reusable component:
  - Make a template ("Pattern Wizard")
The Pattern Wizard

- A template-code generator
- Used as example code
  - Each pattern is presented and exemplified
  - Main design of the Pattern Wizard rewritten to use the Design Pattern in question
- Partially implemented in Eiffel
The Pattern Wizard (2)

- Three main parts:
  - A GUI generated from reading a DTD-file (e.g. by reading “abstract_factory.dtd”)
  - An XML-file generator (e.g. generating “abstract_factory.xml”)
  - *Implemented*: A code-generator (e.g. generating framework code for the Abstract Factory Pattern by reading “abstract_factory.xml”)
The Pattern Wizard (3)

- Pending the release of Eiffel Studio for Mac, the presentation will now briefly change its platform.. :-/
- ...
- We're back! :-)

Turning Design Patterns into REUSABLE COMPONENTS
Pattern Wizard

Examples

• Each example take use of the same product classes:
  • Abstract family-products:
    • Classes, non-instantiable classes, routines
  • Concrete family products:
    • Eiffel: Classes, deferred classes, features
    • Java: Classes, abstract classes, methods

• These classes are instantiated or created according to the (creational) Design Pattern in question
Abstract Factory

• Group related objects in “abstract families” and sub-class these families to provide a concrete implementation
• The Pattern Wizard example:
  • Class `ABSTRACT_FACTORY` defines an interface for manufacturing products
  • Sub-classed by `EIFFEL_FACTORY` and `JAVA_FACTORY`
Abstract Factory (2)

• Applicability as reusable component

• Problems:
  • We don't know in advance how many products there are
  • We don't know the input parameters of the products constructors
  • The abstract factory (who's objects manufacture products) needs a manufacture method for each product type

• Conclusion:
  • Had no successful attempt implementing it as a reusable component
  • A template was made
Factory method

- A “simple” version of the Abstract Factory
- Has a deferred factory method creating products
  - Move the functionality for creating products within the “client”-class
  - Useful when one or few product families exist
Factory Method (2)

• Pattern Wizard example
  • Have a deferred client containing all functionality but the instantiation of CLASS- and METHOD -objects
• Sub-class the client for each programming language supported (e.g. EIFFEL_CLIENT, JAVA_CLIENT etc.)
Factory Method (3)

• Applicability as reusable component

  • Problem:
    • We don't know in advance how many products there are
    • We don't know the input parameters of the products constructors

  • Conclusion:
    • Had no successful attempt implementing a reusable component
    • Made a template instead
Builder

- Constructs complex objects (objects containing several products)
- Applicable when the sequence of the object construction is important
- `create_beer` feature must be called after `create_glass` etc.
Builder (2)

- Pattern Wizard example
  - The creation of a Class-object can be regarded as a construction process
    - Instantiate the method-objects therein
    - Instantiate other objects (collaboration-objects, invariant-objects etc.)
  - By implementing it as a Builder, the topology of the construction process is maintained regardless of the programming language to generate code to
Builder (3)

• Applicability as reusable component
  • Problems:
    • We don't know how many product classes there are
    • Would need a build method for each product
    • We neither can nor does it make sense to define a “generic” number of methods
  • Conclusion:
    • Not applicable as reusable component
Prototype

• Specify what objects to create using a prototypical instance and create new objects by copying this prototype
• Can be regarded as a flexible version of the Abstract Factory
• Allows a run-time construction of product families
Prototype (2)

- Easy to implement in Eiffel using the `clone` or `deep_clone` features
- Pattern Wizard example
  - Find the target programming language and instantiate its corresponding Class- and Method objects to be used as prototypes
  - Clone these objects as they are needed, only “seeing” the interfaces defined in their deferred classes
Prototype (3)

• Applicability as reusable component
  • The clone and deep_clone features are available for any class

• Conclusion
  • Applicable as reusable component
Prototype (4)

• Made the reusable class PROTOTYPE
• One PROTOTYPE class is declared for each product class (by taking the product-class as a generic parameter)
• After instantiating a prototype object it is sent as parameter to the constructor of the PROTOTYPE class
Singleton

• Ensure a class only has one instance and provide a global point of access to it
• E.g. “There can be only one”
  • An accounting system should serve only one company
Singleton (2)

• Pattern Wizard example
  • Make sure there is only one object creating the products
  • Use the Singleton pattern to implement the functionality for creating a factory object
  • The factory object was created within a once feature:
    • The same factory object will be returned each time this feature is called
Singleton (3)

- Applicability as reusable component
  - Problem:
    - No static methods in Eiffel
    - A once feature is once per system; not once per class
    - By using invariants to verify that an instance object indeed is a Singleton (there is only one), a Singleton class cannot be sub-classed
  - Conclusion
    - Not applicable as a reusable component
Conclusion

• Abstract Factory, Factory Method and Builder:
  • Made templates

• Prototype:
  • Made a reusable component

• Singleton:
  • Not trivial to implement in Eiffel, made a template
Evaluation

• Problems:
  • Needed some time porting my head from Java to Eiffel
  • Had some fights with Eiffel Studio
• !Problems:
  • Very helpful people
  • Learned a lot (Eiffel, Design Patterns)
  • Got to try raclette
Takk for oppmerksomheten
Danke für die Aufmerksamkeit
Merci du votre attention