Frameworks and Patterns - Framework Extension Patterns

Prof. Dr. U. Aßmann
Software Engineering Group
Faculty of Informatics
Dresden University of Technology
Version 07-1.0, Jan 08, 2008
Literature (To Be Read)

- E. Gamma. The Extension Objects Pattern. Conf. On Pattern Languages of Programming (PLOP) 97, ACM. [Link](http://portal.acm.org/citation.cfm?id=273448.273455)


Further Literature


Goal

► Studying extensible framework hook patterns
  ▪ Understand patterns Extensions Object, Role Object, and Genvoca
  ▪ See how layered frameworks can be implemented by Role Object and Genvoca

► Understand these patterns as extension points of frameworks, i.e., framework hook patterns
Frameworks Must Be Extensible

- Frameworks must evolve, be adapted
- Idea: instead of variability hooks, use extensibility hooks
  - based on basic extensibility patterns
- Presented in this lecture:
  - Gamma's Extension Object Pattern (EOP)
  - Layered frameworks
    - Riehle/Züllighoven's RoleObject pattern (ROP)
    - Batory's mixin layer pattern (GenVoca pattern)
The ExtensionObjects Pattern (EOP)
Structure of ExtensionObjects

- Whenever a complex object has non-mandatory parts that can be added, if necessary
- *Extension* is the base class of all extensions
- *AbstractExtension* defines an interface for a concrete hierarchy of extension objects
- Extensions can be added, retrieved, and removed by clients
Example: Spellcheckers in Document Models

- E.g., OpenDoc or OLE documents

```
<<template>>
Document

getExtension(name)
addExtension(name, Extension)
removeExtension(name)

ConcreteDocument

getExtension(name)
addExtension(name, Extension)
removeExtension(name)

Client

<<hook>>
Extension

owner

Spellchecker

1 or n
extension

WordCounter

French Spellchecker
English Spellchecker
German Spellchecker
```
Discussion of EOP

► If there is 1 extension object, naming is not necessary
► If there are n extension objects, a dictionary (map) has to map names to extension objects
► Advantages
  - Complex objects can be split into simpler parts
  - Extensions can model (optional) roles of objects
  - Extensions can be added dynamically and unforeseen
► Disadvantage
  - Clients have to manage extension objects themselves, and hence, are more complex
  - Extension objects suffer from the object schizophrenia problem: *this* of an extension object is the subject..
► Relations to Other Patterns
  - If many objects of an application have the same roles that are realized by extension objects, ExtensionObjects can be generalized to the Role Object Pattern
Since with EOP, clients have to manage extensions themselves, the use of the template object in the framework does not help to use the hook objects.
EOP as Framework Hook Pattern

- Since the hook object is not mandatory, also 1-H=T is a real extensibility pattern for frameworks.

1-H=T  
T has 1 H part  
T owns H

n-H=T  
T has n H parts  
T owns H parts

T, H

T, H

T = H  

T*, H

T*, H
Optional Tools for Documents in an Office Framework

Office Framework

* Document
  = SpellChecker
  = WordCount
  a.s.o
Extensibility of Frameworks with Layers

... with Layered Role Object Frameworks
Case Study GEBOS

- GEBOS is a banking application for RWG banking group with 450 banks, south of Germany
  - Banking applications, with services: tellers, loans, stocks, investment, self-service
  - 2500 C++ classes, arranged in frameworks, Arranged in layers

- Concepts of the bank application domain
  - Banks organize themselves in **business sections** (tellers, loans, etc.)
    - Department of specialists that have a certain expertise (loans, teller, investment)
  - **Workplace contexts**
    - Service centers offer customers an all-in-one service
    - Services of the business sections
    - Every workplace needs different application systems
  - **Business domain**
    - Business objects such as bill, order, account
Gebos demonstrates that it is advantageous to structure an application framework into layers:

- Application layers, Business Section layers, Business domain layers
- Desktop Layer, Technical kernel layer
Layers

- Technical Kernel Layer
  - Service layer, independent of other layers
  - Domain independent, application independent
  - Is a framework itself
    - Collections
    - Middleware
    - Wrappers
    - Garbage collection, late creation, factories, trace support
  - Is a blackbox framework

- Desktop Layer
  - Support for interactive workplaces
  - Contains a tool construction framework (for the Tools&Materials approach)
  - MVC framework, Folder framework, Value framework for business and domain values
    - AccountNumber, clientNumber, Money etc
  - Look and feel, reusable for office domains with GUI applications
Layers

- **Business Domain Layer** contains the business core concepts: Account, customer, product, value types
  - Shares knowledge for all business sections
  - Think about how to divide the knowledge between business domain layer and business section layers

- **Business Section Layers**
  - Subclassing business domain and desktop layers, “inherits” knowledge from both
  - Business section concepts: Borrower, investor, garantor, loan, loan account, tools. Organizational entities and notions
  - Distinguish from business domain

- **Application Layers**
  - Application concepts
  - Separate from Business Sections, because workplaces need different functionality from different business sections
  - Uses (and inherits) from all other layers
Goals in Framework Design of GEBOS

- Minimize coupling between frameworks and application systems
  - Frameworks should never be touched when developing an application system
- Model different facets of business sections, products, and business domain concepts
  - Use role-object design pattern
- Minimize coupling between the layers
  - Separate concepts from implementation
  - Move implementation to lower layers
- Achieved with the RoleObject pattern
The RoleObject Pattern
The Role-Object Pattern (ROP) is both a variability and extensibility pattern

- Realizes the “dispatch on all layers” for application frameworks
- Can easily be extended with new layers

Extension of a core layer (a blackbox framework of core objects) with layers of delegatees (role objects)

- A conceptual object of the application is split over all layers
- Core and role objects conceptually belong together to the conceptual object, but distribute over the layers
- Role objects are views on the conceptual object
Riehle/Züllighovens Role Object Pattern (ROP)
Role Object Pattern with Inheritance Drawn Upwards

Business Domain Layer
- Customer
  - CustomerCore
    - CustomerRole

Business Section Layers
- Borrower
- Guarantor
- Investor

Application Layer
- Client

<<interfaces>>
<<template classes>>

<<implementations>>
Runtime Behavior

- At runtime, RoleObjects pass queries on to the CoreObjects
- The core knows all RoleObjects, and distributes requests (Mediator)
Riehle/Züllighovens Role Object Pattern Abstracted

Application

Client

Core Layer 1

Conceptual

Core

Role

Layer 2

Role1

Role2

Role3

<<implementations>>

Layer 3

Role1

Role2

Role3

<<implementations>>
Riehle/Züllighovens Role Object Pattern Variant 2

Application

Core Layer 1

Core

Conceptual

Role

Layer 2

Role1

Role2

Role3

Layer 3

Role1

Role2

Role3

<<interfaces>>

<<implementations>>
Different Role Objects May Play Same Role Type

- Over time, the role object for a role type may change, due to polymorphic behavior of the role
- This expresses states of the role type in the application
  - E.g., Borrower --> UnsafeBorrower --> TrustedBorrower
RoleObject and Other Patterns

► Role object pattern is not a Composite
  ▪ It is based on n-H<=T, i.e., n-ObjectRecursion and Composite
  ▪ All role objects inherit from the abstractum, but no recursive part-of relationship
  ▪ Remember, n-ObjectRecursion based patterns lend themselves to extension, others only to variability
  ▪ And n-H<=T framework hook patterns provide extensible frameworks

► Role object pattern is not a Decorator
  ▪ 1:n relationship between core and role objects
  ▪ Not between core and abstractum

► However, the relation between RoleObject and CoreObject is a Decorator (in variant 1)
  ▪ Role objects decorate the core object, and pass requests on to it
Prof. Uwe Aßmann, Design Patterns and Frameworks

**Role Object Pattern Vs Inheritance**

---

**Business Domain Layer**

- Customer
  - CustomerCore
  - CustomerRole

---

**Business Section Investment Layer**

- Investor
  - Savings Account
  - Special Term Savings

---

**Business Section Loan Layer**

- Borrower
  - Loan Account
  - Special Term Loan

---

<<template classes>>
Role Object Pattern Vs Inheritance

Business Domain Layer

- Customer
  - CustomerCore
  - CustomerRole

- Account
  - AccountCore
  - AccountRole

- <<interfaces>>

Business Section Investment Layer

- Investor
- Savings Account

Business Section Loan Layer

- Borrower
- Loan Account
Comparison of Role Objects with Inheritance

► Simple inheritance has one instance of a subclass at a time
  ▪ Subclass can change over time (polymorphism)

► The role object has many of them at the same time
  ▪ All role objects can change (role polymorphism)

► Only changes in the base layers (technical, presentation, business) affect other layers
  ▪ Changes in the business section layers do not affect the business domain layers

► The relation of core and role objects is a special form of part-of (combined with inheritance)
Role Object Pattern As Template and Hooks

Business Domain Layer

- <<template class>> Customer
  - <<template class>> CustomerCore
  - * <<hook class>> CustomerRole

- <<template class>> Account
  - <<template class>> AccountCore
  - <<hook class>> AccountRole

Business Section Investment Layer

- Investor
- Savings Account

Business Section Loan Layer

- Borrower
- Loan Account
Riehle/Züllighovens Layer Pattern As Framework Hook Pattern

Conceptual

Role

Core

T2 has H parts

H and T2 inherit from T1

Core-Role-Pattern

n-TrH mini-connector

Special partOf
RoleObject Ensures Extensibility

► The RoleObject pattern lends itself not only to variability, but also to static and dynamic extensibility
  ▪ If a framework hook is a role object pattern, the hook can be extended in unforeseen ways **without** changing the framework!
  ▪ New layers of the application or the framework can be added at design time or runtime

► Powerful extension concept
  ▪ Whenever you have to design something complex which should be extensible in unforeseen ways, consider Role Object
Riehle/Züllighovens Layered Role Object Framework
Extension in Layered Role Object Frameworks

Core Layer

Role Layer 1

Role Layer 2

Role Layer 3

Role Layer 4
RoleObject Can Implement Dimensions That Are Not Independent

- The role objects implement dimensions
  - Core object implements primary dimension
  - Role object secondary dimension
- Role objects realize *one conceptual object*, instead of a role model crosscutting several conceptual objects
- Facets are independent dimensions of a conceptual objects
  - Every dimension can be varied independently
- Comparison to the standard implementation of facets by n-Bridge (see Chapter “Simple Extensibility”)
  - n-Bridge has no inheritance between ConceptualObject, Core and Role
  - Calls to the role are not dispatched to the LogicalObject
  - Bridges must not inherit from each other, RoleObjects can
Benefit of Layered Role Objects Frameworks

- Implements conceptual objects with layered dependent dimensions
  - Not only independent dimensions
- Together with layering,
  - Easily extensible
  - Enormous variability
  - Simple structure for extensible product line architecture results
Layered Frameworks for Business Software

► Dispatch on all layers is necessary
  ▪ Implementation without multimethods (in standard languages) very hard. Only CLOS, Cecil, and MultiJava are good here
  ▪ If generics are available, one hierarchy can be flattened by generic expansion (better, see GenericTemplateClass pattern)

► That is one reason why business frameworks are so hard
  ▪ SanFrancisco business framework of IBM didn't make it though a dynamic extensibility pattern
  ▪ That's also why these applications are so expensive
The JWAM Framework

... as an example for a layered framework
Java WAM (Werkzeug Automat Material) is a framework for the Tools&Material pattern language [www.jwam.de](http://www.jwam.de).

The JWAM site has a lot of interesting papers, e.g., the PhD thesis of Bäumer.
JWAM has a Kernel

- 100 classes and interfaces
- Simple applications can be built with the kernel only
- Extensions can be added, extension components:
  - Equipment components
    - Ready to use packages such as desktop, registry, form-service
  - Integration components
    - Database connection...
The GenVoca Pattern, Mixin Layers, and Layered Mixin Frameworks
The Mixin Concept

- A Mixin is an extension of a class.
- Mixins can be expressed as class fragments that can be parameterized with a superclass.
- Mixins can implement roles and facets.

```cpp
template <class S>
class EmployeeMixin extends S {
    // class extension..
    Salary salary;
    Employer emp;
}
```

```cpp
EmployeeMixin<Person> employeeOfPerson;
EmployeeMixin<German> employeeOfGerman;
EmployeeMixin<Club> employeeOfClub;
```

Prof. Uwe Alßmann, Design Patterns and Frameworks
The GenVoca Pattern

- If several mixin parameterizations are nested, the GenVoca pattern results [Batory]

```cpp
template <class S> class EmployeeMixin extends S {
    Salary salary;
    Employer emp;
}
template <class S> class ParentMixin extends S {
    Child child;
    Money kindergeld;
}
template <class S> class HobbyMixin extends S {
    Hobby hobby;
}
// Persons composed with GenVoca pattern
HobbyMixin<ParentMixin<EmployeeMixin<Person>>> assmann;
EmployeeMixin<ParentMixin<HobbyMixin<Person>>> assmann2;
// Have assmann and assmann2 the same type?
```
GenVoca Variations

- When different variants exist for a “abstraction layer”, parameterizations express configurations of a product line

```cpp
// Variants
Person: Man, Woman
ParentMixin: FatherMixin, MotherMixin
EmployeeMixin: TimedEmployee, PermanentEmployee
HobbyMixin: PigeonFriend, Sportsman, GolfPlayer

// Compositions
GolfPlayer<TimedEmployee<Father<Man>>> assmann;
PigeonFriend<PermanentEmployee<Father<Man>>> miller;
GolfPlayer<PermanentEmployee<Mother<Woman>>> brown;
```
Variations on Different Abstraction Layers form Product Variants

Person

<<parameterize>>

ParentMixin

<<parameterize>>

EmployeeMixin

<<parameterize>>

HobbyMixin

Man

Father

TimedEmployee

PermanentEmployee

GolfPlayer

PermanentEmployee

Woman

Mother

GolfPlayer

PigeonFriend
Variations on Different Role Layers

- In fact, abstraction layers correspond to role layers of logical objects
- Roles *collaborate*, but are not implemented by role objects, but by mixins
Discussion

- A *mixin layer* groups all mixins of a role abstraction layer
- Mixins play in the GenVoca pattern the same role as role objects in the role object pattern and layered role frameworks
  - However, all role objects are *embedded* into one physical object
  - There is a physical identity for the entire logical object
  - No object schizophrenia to be avoided
  - GenVoca applications are more efficient, since they merge all roles together into one physical object (see the Aßmann's law on role merging)
- Similarly to layered role object frameworks, layered GenVoca frameworks can model big product lines
  - Every abstraction layer (mixin layer) expresses variability
  - New mixin layers model extensibility
The Mixin Layer Pattern

While the GenVoca pattern deals with single stacking of parameterizations, the MixinLayer pattern groups all roles of an abstraction layer together and composes entire layers.

MixinLayer treats all logical objects of an application.
A mixin layer gets a name and can be exchanged consistently for a variant, changing the behavior of the entire layer.
Composition of Mixin Layers

- Mixin layers are composed similarly to single GenVoca mixins
  - Meaning: All role classes are consistently exchanged with their layer

CoreLayer: FullTime, PartTime
ParentLayer: FatherLayer, MotherLayer
EmployeeLayer: Deliberate, ...
HobbyLayer: WorkAsHobby, Slave....

// This is now mixin layer composition!
WorkAsHobby<Deliberate<FatherLayer<FullTime>>> assmann;
The role classes of upper layers form super classes of the layer class.

The following pattern allows for separate parameterization of all role mixins, not the layer as a whole.

class Layer <class Super, class RoleSuper\(_1\), .., class RoleSuper\(_n\)>
   extends Super {
   class Role\(_1\) extends RoleSuper\(_1\) { .. }
   ..
   class Role\(_n\) extends RoleSuper\(_n\) { .. }
   .. additional classes..
}
Implementation of Mixin Layers with Designated Inner Classes

- If the target language permits to have inner classes that can be designated by an expression, mixin layers can be inherited as a whole.
- The super mixin layer can be selected by one single expression $L<L_1>$.

```java
class Layer <class Super>
// The class Super has n inner role classes RoleSuper_1, .., RoleSuper_n
extends Super {
    class Role_1 extends Super.RoleSuper_1 { .. }
    ..
    class Role_n extends Super.RoleSuper_n { .. }
    .. additional classes..
}
```
Example: A Graph Framework

- Graph applications can be structured into mixin layers
- ConnectedOnDFTUndirected = CRL1<CL1<VN1<TL1<RL1>>>>
- ConnectedOnBFTRelDirected = CRL1<CL1<VN2<TL2<RL2>>>>

Diagram:
- Representation Layer:
  - RL1: GraphUndirected
  - RL2: GraphDirected

- Traversal Layer:
  - TL1: DFT
  - TL2: BFT

- Vertex Numbering Layer:
  - VN1: VertexNumber
  - VN2: VertexReverseNumber

- Cycle Layer:
  - CL1: GraphCycleCheck

- Connected Region Layer:
  - CRL1: GraphConnected
Layered Mixin Frameworks vs Layered Role Object Frameworks

- Every mixin layer corresponds to a role layer
- Mixin layers form *frameworks* that can be extended by mixin layer composition towards applications
- Same variability effects for big product lines

![Diagram showing layered mixin frameworks vs layered role object frameworks](image-url)
Layered Mixin Frameworks vs Layered Role Object Frameworks

- Unfortunately, the direction of generality is usually drawn in the opposite way in mixin layer frameworks and role object frameworks.
- If we agree to put the “most general abstraction layer” downmost, the dependencies go into the same direction.
- Features on the upper layers depend on the lower layers.

Diagram:

- Application Layers:
  - Adviser
  - Desktop
  - Telephone
  - Banking
  - Teller

- Business Section Layers:
  - Teller
  - Loans
  - Investment

- Business Domain Layer:
  - Account
  - Loan
  - Product
Layered Mixin Frameworks vs Layered Role Object Frameworks

► Essentially, layered role object frameworks and layered mixin frameworks provide the same concept for variability and extensibility

► Difference: mini-connector
  - Layered role object frameworks use as mini-connector the Role Object Pattern
  - Layered mixin frameworks use as mini-connector the GenVoca pattern
How To Find Concerns for a Layered Framework

Example: Layered Frameworks for the Semantic Web
A New Application Area: Semantic Web Applications

► Semantic web:
  ▪ Standardization technology for the Web and many application domains
  ▪ Definition of *ontologies*, standard dictionaries
  ▪ Based on inheritance and constraints

► Every application domain will have its “Semantic Web ontology”

► How to build product families for those domains?
The Concerns of an Application in the Semantic Web

- Which concerns exist?
- After a little thought: three groups of concerns. (This is not complete, there might be more)
Layered Frameworks for Product Lines on the Semantic Web

- We can sort the acyclically dependent concerns into a layered architecture, in which several RoleObjects crosscut all layers
  - On every level, there are constraints to check the layer for consistency
  - All role objects on the layer are checked by the constraints

<table>
<thead>
<tr>
<th>Application Layers</th>
<th>Architectural Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Layer (Personalization)</td>
<td>Application Family Layer</td>
</tr>
<tr>
<td>Application Layer</td>
<td>Architectural Style Layer</td>
</tr>
<tr>
<td></td>
<td>Concrete Component Model Layer</td>
</tr>
<tr>
<td>Core Conceptual Layers</td>
<td></td>
</tr>
<tr>
<td>Component Language Layer</td>
<td></td>
</tr>
<tr>
<td>Composition Time Layer</td>
<td></td>
</tr>
<tr>
<td>Generic Component Layer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Framework</th>
<th>Component</th>
<th>VariationPoint</th>
<th>Composition operator</th>
<th>Constraint checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concern Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concern Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concern Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concern Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concern Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concern Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concern Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prof. Uwe Aßmann, Design Patterns and Frameworks
Layers can be Instantiated Differently

- On every layer of the layered framework, there is variation and extensibility
- New user constraints
- New application constraints
- New application family constraints
- New architectural constraints
- New component models
- New component languages
Different Languages in One Framework

- Since the language is a layer, it can be exchanged
- Several ontology languages can be used for components in Semantic Web applications
  - Datalog, Prolog, OWL
Application Layers

User Layer (Personalization)  Uwe Assmann Banking Portfolio

Application Layer  MoneyDeposit Inc. Web Services

Architectural Layers

Application Family Layer  Banking Transaction-based Web Services

Architectural Style Layer  Transaction-based Web Services Model

Concrete Component Model Layer  Web Services Model

Core Conceptual Layers

Component Language Layer  Datalog, Prolog, OWL

Composition Time Layer  Fragment Component

Generic Component Layer  Component
Different Architectures are Possible for One Component Model

► Since the architectural styles can be exchanged for the same component model
Different Component Models Can Coexist

- Interoperability of Semantic Web application is simplified
Layered Frameworks and Component Models

► Once, if languages and component models are layers, layered frameworks can be generalized considerably.
  - Implementation with Layered ROP frameworks or Layered mixin frameworks

► It becomes possible to build totally heterogeneous applications:
  - Different framework and component languages
  - Different architectures and architectural styles
  - Different product lines (application families)

► This is not reality yet, this is a dream
  - There is only a design so far
  - But we going towards it
What Have We Learned?

► How can we structure a Product Line as Layered Framework?
  ▪ ExtensionObjects is a simple extension mechanism for frameworks
  ▪ Layered frameworks provide variability and extensibility for thousands of different products in a product line

► Process for layered frameworks:
  ▪ Identify concerns (abstraction layers), which crosscut all or many objects. These concerns are similar to facets, but not independent
  ▪ Sort them according to their (acyclic) dependencies
  ▪ Use ROP or Genvoca pattern for implementation
  ▪ Use framework role layers or mixin layers for a layered application
The End