13. Frameworks and Patterns - Framework Extension Patterns

Prof. Dr. U. Aßmann  
Software Engineering Group  
Department of Computer Science  
Technische Universität Dresden  
Version WS14/15 - 12/15/14  

**Lecturer:** Dr. Sebastian Götz

1) Extension Object Pattern  
2) Large Layered Frameworks  
3) Role Object Pattern  
4) GenVoca Pattern  
5) Mixin Layer Pattern
Literature (To Be Read)


Further Literature


- JWAM sites
  - http://www.c1-wps.de/forschung-und-lehre/fachpublikationen/
  - www.jwam.de
  - http://sourceforge.net/projects/jwamtoolconstr/


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)
13.1 The ExtensionObjects Pattern (EOP)

Extensions of Objects, visible for the Client
Structure of Extension Objects

- 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

```
Subject

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

ConcreteSubject

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

Extension

owner

1 or n
extension

AbstractExtension

Concrete Extension1
Concrete Extension2
Concrete Extension3
```
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)

<<hook>>
Spellchecker

owner

Extension

1 or n
extension

WordCounter

French Spellchecker
English Spellchecker
German Spellchecker

Client
```
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: the logical *this* of an extension object is the subject, but the physical *this* is the extension object

**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.
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
Optional Tools for Documents in an Office Framework

Office Framework

Document

* SpellChecker

WordCount

a.s.o
13.2 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
  - **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, guarantor, 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
13.3 The RoleObject Pattern
Framework Extensibility with Riehles Role-Object Layers

- 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 (complex object, subject)** 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/Züllighoven's Role Object Pattern (ROP)

Application Layers

- Client

Business Section Layers

- Borrower
- Guarantor
- Investor

Business Domain Layer

- Customer

CustomerCore

CustomerRole

Static knowledge

Potential run-time access
Role Object Pattern with Inheritance Drawn Upwards
Run-Time Structure: Deep Roles

- At runtime, RoleObjects pass service requests (queries) on to the core
  - RoleObjects can be stacked in a Decorator chain (deep roles)
- The core knows all RoleObjects, and distributes requests (Mediator)
  - The core manages the RoleObjects in a map that can be dynamically extended

Diagram:

- At runtime, RoleObjects pass service requests (queries) on to the core
- RoleObjects can be stacked in a Decorator chain (deep roles)
- The core knows all RoleObjects, and distributes requests (Mediator)
- The core manages the RoleObjects in a map that can be dynamically extended
Riehle/Züllighoven's Role Object Pattern Abstracted ("Deep Roles")
Riehle/Züllighoven's Role Object Pattern
Variant 2 ("Flat Roles")

- Variant 2 has no Decorator; roles only know cores
Variant “Flat Roles”: Run-Time Structure

- At runtime, RoleObjects pass service requests (queries) on to the core
  - RoleObjects can be stacked in a Decorator chain
- The core knows all RoleObjects, and distributes requests (Mediator)
  - The core manages the RoleObjects in a map that can be dynamically extended
- At runtime, RoleObjects pass service requests (queries) on to the core
  - RoleObjects are directly linked to the core (flat roles)
Run-time Behavior of ROP

- Change of role:
  - Different Role Objects may belong to the same role type (e.g., working for multiple companies)
  - Over time, the role object for a player may change
  - This expresses states of the player in the application
    - E.g., Borrower --> UnsafeBorrower --> TrustedBorrower

- Roles are created on-demand
  - In the beginning, the Subject is slim, i.e., carries no roles.
  - At service requests, the core creates roles and enters them in the role map
Variant 3: Core Layer with Traded Call

- To add services dynamically (beyond the service interfaces in the conceptual object), add a *trader* to the core
  - A **trader** is a method that interprets a service request based on a service description

```
// enter traded role //
Interpret serviceDescription;
add concrete role and service under service description
in role map;
```

```
// Traded call //
Interpret serviceDescription;
Lookup concrete service in role map;
Call;
```
RoleObjectPattern and Other Patterns

- ROP is not only a Decorator
  - It is based on 1-H<=T, i.e., 1-ObjectRecursion
    - All role objects inherit from the abstractum
  - Remember, 1-ObjectRecursion based patterns lend themselves to extension
  - And 1-H<=T framework hook patterns provide extensible frameworks
  - 1:n relationship between core and role objects
  - Role objects decorate the core object, and pass requests on to it
Role Object Pattern Vs Inheritance (White-Box Framework Layers)

**Business Domain Layer**
- **Customer**
- **CustomerCore**
- **CustomerRole**

**CustomerCore** has a relationship with **CustomerRole** (denoted by an asterisk). **CustomerRole** has a relationship with **Account** and **Product**.

**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 (White-Box Framework Layers)

Business Domain Layer

Customer

CustomerCore

*CustomerRole

<<interfaces>>

Account

AccountCore

*AccountRole

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 with Template and Hook Stereotypes

Business Domain Layer

- **Customer**
  - **CustomerCore**
  - **CustomerRole**

- **Account**
  - **AccountCore**
  - **AccountRole**

Business Section Investment Layer

- **Investor**
- **Savings Account**

Business Section Loan Layer

- **Borrower**
- **Loan Account**
Role Object Pattern and Role Models on Role Layers

- Usually, roles of one subject talk to other roles of another subject on the same layer (within a role model)
- Cores never talk to each other directly

![Diagram of role objects and models across different layers](image-url)
Switching Variable Role Layers

- At run time, entire role models on role layers can be exchanged (variable role layers)

![Diagram showing role layers and relationships between entities like Customer, Account, Garantor, Garantee Contract, Borrower, and Loan Account. The diagram illustrates how different layers interact and exchange roles at runtime.](image-url)
Riehle/Züllighoven's Layer Pattern As Framework Hook Pattern

n-TrH
- T2 has H parts
- H and T2 inherit from T1
- H knows T1

Core-Role-Pattern

n-TrH mini-connector

Special partOf
ROP Ensures Extensibility

- The ROP lends itself not only to variability, but also to **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 with ROP-Trader
  - Whenever you have to design something complex which should be **extensible in unforeseen ways**, consider Role Object
Riehle/Züllighovens Layered Role Object Framework

Role Layer 1

Role Layer 2

Role Layer 3

Core Layer
Extension in Layered Role Object Frameworks

Core Layer

Role Layer 1

Role Layer 2

Role Layer 3

Role Layer 4
ROP 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 object
  - Every dimension can be varied independently

- Comparison to the standard implementation of facets by Multi-Bridge (see Chapter “Simple Extensibility”)
  - Multi-Bridge has no inheritance between ConceptualObject, Core and Role
  - Multi-Bridge suffers from object schizophrenia, ROP can implement “this()” on itself without object schizophrenia
  - Calls to the role are not dispatched to the LogicalObject
  - Bridges must not inherit from each other, RoleObjects can
Benefit of Layered ROP 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

- For instance: 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
  - 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
13.4 The GenVoca Pattern, Mixin Layers, and Layered Mixin Frameworks
The Mixin Concept

- A **mixin** is a partial class, for an extension of another class
- A **mixin-base** is a class with a generic super class, a mixin parameterizes this
- Some languages have mixins (Scala, C#, Eiffel); otherwise, mixins can be expressed as class fragments that can be parameterized with a superclass (C++)
- Mixins can implement (static) roles and facets

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

```
EmployeeMixin<Person> employeeOfPerson;
EmployeeMixin<German> employeeOfGerman;
EmployeeMixin<Club> employeeOfClub;
```
The GenVoca Pattern

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

```cpp
// Persons composed with GenVoca pattern
HobbyMixin<ParentMixin<EmployeeMixin<Person>>> assmann;
EmployeeMixin<ParentMixin<HobbyMixin<Person>>> assmann2;

// Have assmann and assmann2 the same type?
```
When different variants exist for an “abstraction layer”, parameterizations express configurations of a product line

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

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

Variants can be formed on every layer
Variations on Different Role Layers

- Abstraction layers correspond to *role layers* of complex objects
- Roles *collaborate*, but are not implemented by role objects, but by mixins

Diagram:

```
  Person <<parameterize>> ParentMixin
  Man            Father
  Woman          Mother

  <<parameterize>>
  EmployeeMixin <<parameterize>> TimedEmployee <<parameterize>> Permanent Employee

  <<parameterize>>
  HobbyMixin <<parameterize>> GolfPlayer <<parameterize>> PigeonFriend
```
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)
    - **But**: only static extensibility!
- 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
13.5 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;
```
Implementation of Mixin Layers with GenVoca Pattern and Inner Classes

- 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

```java
class Layer <class Super, class RoleSuper₁, .., class RoleSuperₙ> extends Super {
class Role₁ extends RoleSuper₁ { .. }
.. 
class Roleₙ extends RoleSuperₙ { .. }
.. 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 \( L<L1> \)

```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 [Herrejon Batory]

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

```
<table>
<thead>
<tr>
<th>Person</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParentMixin</td>
<td>ParentalWork</td>
</tr>
<tr>
<td>EmployeeMixin</td>
<td>WorkRights</td>
</tr>
<tr>
<td>HobbyMixin</td>
<td>LeisureWork</td>
</tr>
</tbody>
</table>
```

```
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

- 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:

- Person → ParentMixin → EmployeeMixin → HobbyMixin
- Work → ParentalWork → WorkRights
- LeisureWork

Layers:

- **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: dynamic extensibility
  - Layered role object frameworks allow for it
  - Layered mixin frameworks don't
What Have We Learned?

► Extension Objects Pattern
► Extensible Framework Hook Patterns
  ▪ Using Role Object Pattern
  ▪ Using Genvoca (MixinLayers)
► Role Object Pattern for dynamic extensibility
► Genvoca for static extensibility