Role-based Design –
A Concept for Understanding Design Patterns and Frameworks

Prof. Dr. U. Aßmann
Chair for Software Engineering
Faculty of Informatics
Dresden University of Technology
Literature (To Be Read)

Other Literature


Other Literature


► Walter Zimmer. Relationships Between Design Patterns. Pattern Languages of Program Design 1 (PLOP), Addison-Wesley 1994
Goal

- Understand the difference between roles and objects, role types and classes
- Understand role mapping to classes
  - How roles can be implemented
- Understand role model composition
- Understand design patterns as role models, merged into class models
- Understand composite design patterns
  - Understand how to mine composite design patterns
- Understand role types as semantically non-rigid founded types
- Understand layered frameworks as role models
- Understand how to optimize layered frameworks and design patterns
Role-based Design With Role Models
Purpose of Teaching Role-based Design

- Design patterns – although not described as such in [Gamma] – rely on the concept of **roles**
- A design pattern must be *matched* in an application,
  - i.e., there must be some classes in the application that play the roles of the classes in the design pattern.
  - Every class in the design pattern is a *role type*
  - The attached class of the application plays the *role* of the class in the design pattern
What are Roles and Role Types?

- Roles: *views* onto an object
  - The view can change dynamically
- Hence, a role of an object belongs to an area of concern
- Roles are *played* by the objects (the object is the *player* of the role)
  - Roles are states of an object

![Diagram](Diagram.png)
What are Roles and Role Types?

- Roles are *services* of an object in a context
  - Roles can be connected to each other, just as services are connected to client requests
- Roles are tied to *collaborations*
What are Role Types?

- Role types are *service types* of an object
- Role types are *view types* onto an object
  - The role type can change dynamically
- The role type is a *dynamic type*
  - Role types are state characterizations of an object
  - An object plays a role of a role type for some time
- A role type is a *part of a protocol* of an class
  - A role is often implemented by interfaces
- Problem: often, we apply the word “role” also on the class level, i.e., for a “role type”
- A *role model* is a set of object collaborations described by a set of role types
  - A constraint specification for classes and object collaborations
A Class and Role Type Diagram

- Also called a class-role model
- Roles (oval boxes) are put on top of classes (rectangles)
- The set of role types of a class is called its role type set
  - Any number of roles can be active at a time

Person

- Employee
  - Father
  - Cyclist
- Diabetics
  - Conservative
  - Swede
A Class-Role Model For Figures in a Figure Editor

- A figure can play many roles
- Roles may be qualified by a role model identifier in brackets
- This class-role model is composed out of several simpler role models
Role Models

- Explanation of some role types
  - `FigureHierarchy.Figure`: regular drawing functions
  - `FigureHierarchy.Child`: child in a figure hierarchy
  - `FigureObserver.Subject`: subject of a Observer pattern, for communication among figures
  - `FigureHierarchy.Parent`: parent in a figure hierarchy
  - `IntFigObserver.Subject`: subject of a Observer pattern, for communication among figures
  - `FigureChain.Successor`: successor in a threaded list (chain) of figures
Role Constraints in the Figure Hierarchy Role Model

- Arrows denote constraints between roles (role constraints)

Role use

Role inheritance means "role-implication: \( a < b \) means the object that plays role \( a \) must also play role \( b \)

Exclusion constraint means "role-prohibition: \( a - b \) means the object that plays \( a \) must not play \( b \) and vice versa"
How To Develop Role Models

► Ask the central question:
  - Which role does my object play in this context?
  - Which responsibility does my object have in this context?
  - Which state is my object in in this context?

► If you develop with CRC cards, the questions lead to a grouping of the responsibilities (i.e., roles) on the CRC card
  - Remember: a role model specifies roles of objects in context, i.e., in a specific scenario
  - Keep the role model slim, and start another one for a new scenario
Benefit of Role-based Design

► Roles split a class into smaller pieces
► Roles emphasize collaborations in design, i.e., emphasize the context-dependent parts of classes
► Roles separate concerns (every role type is a concern)
► Role models can be reused independently of classes

► Idea: why not develop with role models?
Role-Based Design with Role Models

- Emphasizes collaboration-based design
  - Starts with an analysis of the collaborations (e.g., with CRC cards)
  - Every partner of a collaboration is a role of an object
  - The role characterizes the protocol (interaction) of the object in a collaboration
Steps In Role-Based Design

► First, do role models
  - Roles are all kept distinct
  - Find out about role constraints that constraint which objects execute which roles

► Secondly, merge role models
  - And set up new constraints between roles of different models

► Thirdly, role models are mapped to a class diagram
  - By merging the roles to objects
  - Respecting the constraints
  - Role models must be “woven” into class models (*role mapping*)
  - As composition step, the merge is applied
  - (Not the distribution, as in AOP)
Composing Role Models
Design Patterns have Role Models

- Observer role model

Observer (FigureObserver) 0..* Subject (FigureObserver)
Composing (Overlaying) Role Models

- Overlaying the FigureHierarchy with the FigureObserver role model

Bidirectional Inheritance means "role-equivalence:\n\[ a \leftrightarrow b \text{ means the object that plays } a \text{ must also play } b \text{ and vice versa} \]
Role Models of Persons in Business Applications
Role Models of Persons in Business Applications

A Retailer must also play the role of a customer (retailers are customers of banks)

AppClient (Person) -> Person (Person) -> Customer (Customer) -> AppClient (Customer)

Client of Customer role model uses customer role of Customer role model

Customer (Retail) -> Retailer (Retail) -> AppClient (Retail)

Customer (Debitor) -> Debitor (Debitor) -> AppClient (Debitor)

Customer (Investor) -> Investor (Investor) -> AppClient (Investor)
Role Models of Persons in Business Applications

AppClient (Person)  
|                     | Person (Person)  
|---------------------|-----------------|----------------|
| Customer (Customer) | Guarantor (Guarantor)  
| Retailer (Retail)   | Employee (Employee)  
| Debitor (Debitor)   | Investor (Investor)  
| AppClient (Retail)  | AppClient (Investor)  

Guarantor never plays the role of a customer
Composing Role Models to Class Diagrams
Composing Role Models To Partial Class Diagrams

- Classes combine roles
  - Classes are composed of role types
  - Roles are dynamic items; classes are static items
  - So, classes group roles to form objects

- Class models combine role models
  - Class models are composed of role models
  - One role model expresses a certain aspect of the class model

- Partial class models:
  - Role types in a role model can be left dangling (open) for further composition
  - The sub-role-models of a composed role model are called its dimensions
  - A partial class model results
  - Then not all roles are associated to classes
Role Models in the Example

► FigureHierarchy: composite figures (with root figure and other types, such as rectangular or class)
► FigureChain: How objects forward client requests up the hierarchy, until it can be handled
► FigureObserver: Observer pattern, for callback communication among clients and figures
► IntFigObserver: Observer pattern, for communication among figures
The Role Mapping Process and Model-Driven Architecture

► A role model is more *platform independent* than a class model
  ▪ The decision which roles are merged into which classes has not been taken and can be reversed
  ▪ We say: roles are *logical*, classes are *physical*

► In MDA, role models are found on a more platform independent level than class models
  ▪ First design a role model
  ▪ Then find a class model by merging roles into classes (mapping roles onto classes, role mapping)
  ▪ Respecting role constraints
  ▪ Usually, several class models are legal
Role Model Mapping is a Task in MDA

Model mappings

- Business model
- Role Model
- Class Model
- Code

Role Model mapping
Role Mapping is Based on Role Constraints

- **Role-equivalent constraint**: strong constraint: same implementation class
- **Role-implication constraint**: weaker, leaves freedom, which physical class implements the roles
  - If implemented by the same class, the class model is stricter than the role model
  - Embedding roles in a class reduces the number of runtime objects, hence more efficient, less object schizophrenia
  - Split classes allow for better exchange of a role at runtime, since only the runtime object needs to be exchanged
- **Role-implication inheritance constraint**: a role-implication constraint, stressing that the source can be mapped to a subclass of the target

![Diagram](image_url)
Computing Physical Objects

- The role mapping process determines, which physical object inherits from which role-interface.
- The role mapping *computes* the physical objects from minimal splits of the logical objects.

Role model mapping:

- **Role Model** (maximally splitted responsibilities of the logical objects)
- **Class Model** (partially overlayed responsibilities, physical objects)
- **Code**
Implementing Roles
Implementation of Roles

- With interfaces
  - Then, code for the interfaces must be written by hand
- With multiple inheritance
  - Then, there are two layers of classes: role classes and standard classes
- With mixin classes
  - Some language allow for composing “mixin” classes into classes
    - CLOS, Scala
    - “include inheritance” (Eiffel, Sather)
  - A role is like a mixin class
  - No code has to be written by hand
With Interfaces

- Then, code for the interfaces must be written by hand
Most Simply, Roles Types Are Implemented By Interfaces

```java
public class Figure extends
    FigureHierarchy.Figure,
    FigureHierarchy.Child,
    Graphics.Client,
    IntFigObserver.Subject,
    FigureObserver.Subject,
    FigureChain.Predecessor
{
    ... implementations of role-interfaces ...
}
```
Embedding With Multiple Inheritance

- Then, there are two layers of classes: role classes and standard classes
- A standard class must inherit from several role classes
- Disadvantage: a standard class can inherit from a role class only once
Embedding With Mixin Classes

- Some languages allow for composing "mixin" classes into classes
  - CLOS, Scala
  - "include inheritance" (Eiffel, Sather)
- A role is like a mixin class
- No code has to be written by hand
Implementation With n-Bridges and Role Objects

- A role object represents only one role
- A role class only one role type
- There is a core object that aggregates all role objects
- Can be an n-Bridge
- Also with “Role Object” pattern (later)
Roles and Facets

- A faceted class is a class with n *role dimensions*
  - Each facet is a role type
  - Role types are independent of each other
Example of Persons in Business Applications
Role Models of Persons

- Client (Person)
- Person (Person)
- Customer (Customer)
- Guarantor (Guarantor)
- Retailer (Retail)
- Debitor (Debitor)
- Client (Retail)
- Client (Debitor)
- Client (Investor)
- Client (Employee)
- Client (Guarantor)
- Client (Investor)
- Client (Employee)
Implementation With Interfaces (or Mixins)
Implementation of Person With n-Bridge (Role Objects)
Example:
Actors, Films, and Directors
Actors, Films, and Directors

- We model actors, directors, producers, and their films.
- Actors have a genre (lover, serious, comedian) and play on a certain media (TV, cinema, Shakespeare).
- Directors and producers have similar attributes.
- Films also.
- Actors have an age (young, medium, old).
Example Role Model for Actors

- **Profession**
  - Actor
  - Director
  - Producer

- **Person**
  - OldAge
  - MiddleAge
  - YoungAge

- **Genre**
  - LatinLover
  - NordicLover
  - Vamp
  - Lover
  - Comedian
  - Serious
  - LatinLover
  - NordicLover
  - Vamp
  - Lover
  - Comedian
  - Serious

- **Media**
  - ShakespeareActor
  - CinemaActor
  - TVActor
  - ActorMedia
  - DirectorMedia
  - ProducerMedia
  - ShakespeareFilm
  - CinemaFilm
  - TVFilm

- **SkakespeareFilm**, **CinemaFilm**, **TVFilm**
There are Many Ways to Implement This Role Model

- With a facet based model, modelling some role models as class hierarchies of a Dimensional Hierarchies model
Very Simple Class Model for Actors and Films

- 4-dimensional model (facets)

- FilmPerson

- Media
  - ShakespeareFilm
  - CinemaFilm
  - TVFilm

- Person
  - Actor
  - Director
  - Producer

- Genre
  - Lover
  - Serious
  - Comedian

- Age
  - Young
  - Middle
  - Old

employs ✓ hasA
Some roles implemented by overlaying, inheritance, delegation, and role objects.
Design Patterns as Role Diagrams
Structure Diagrams of DP are Role Diagrams

- The “participant” section of a GOF pattern is a *role model*
- Roles of Chain of Responsibility:
  - Chain: (successor, predecessor)
  - ChainUse: (Handler, HandlerClient, Tail, TailClient)
Role Diagram of Composite

- Root role is not in the standard pattern description

Diagram:
- NodeClient
- Root
- Node
- Parent
- Child
- RootClient
- Children
- Parent

Connections:
- NodeClient to Node
- RootClient to Root
- Node to Parent
- Parent to Child
- Children from Parent to Child
- Parent from Parent to Parent
- Parent from Child to Parent
- Children from Child to Parent
Core Role Diagrams of Several Patterns

- Many of them are quite similar
Composite Design Patterns with Role Model Composition
Example: Bureaucracy

- A pattern to model organizations that have a tree-like structure (as opposed to matrix organizations)
- Is composed of the role models of Composite, Mediator, Chain, Observer
Example: Bureaucracy

► The *Composite* defines the organizational hierarchy of managers
► The *Mediator* is used to let talk children talk to their siblings (colleague roles) via a parent (mediator role)
► The *Chain* handles requests of clients
  ▪ Every node may handle requests
  ▪ If a node cannot handle a request, it is passed up in the hierarchy (on the path to the root)
► The *Observer* is used to listen to actions of a node
  ▪ If a child node (subject) changes something, its parent (observer) listens and distributes the information accordingly
Role-Class Model of Bureaucracy
Bureaucracy
Role-Class Model of Figures
Application of Bureaucracy

- For all hierarchies
  - Figures in graphic and interactive applications
  - Widgets in GUIs
  - Documents in office systems
  - Piece lists in production management and CAD systems
  - Hierarchical tools in TAM (see later)
Model-View-Controller (MVC)

- From Tyngre Reenskaug and Adele Goldberg
- MVC role model can be composed from the role models of Observer, Strategy, Composite
This Closes a Big Loop

- Remember, Reenskaug developed MVC 1978 with Goldberg, while working on Smalltalk-78 port for Norway
- Starting from his MVC pattern, Reenskaug has invented role-based design
- 1998, Riehle/Gross transferred role-based models to design patterns
- Today, MVC can be explained as composed role models of other design patterns
Riehle-Gross Law On Composite Design Patterns

The role model of a composite design patterns is composed of the role models of their component design patterns.

Consequences

- Complex patterns can be easily split into simpler ones.
- Variants of patterns can more easily be related to each other, e.g., ClassAdapter and ObjectAdapter.
- Template&Hook conceptual pattern can be explained as role model (see next chapter).
Composition of Simple Variability Patterns
The following is an attempt to build up the basic GOF patterns from simple role models

- It is probably not stable

It explains why Strategy is different from Bridge and TemplateClass, etc.
Derived Method

In a class,

- A *kernel method* implements the feature directly on the attributes of the class, calling no other method.
- A *derived method* is implemented by calling only kernel methods.
Derived Method and TemplateMethod

- TemplateMethod is a DerivedMethod that has:
  - an additional template/hook role model
  - Inheritance hierarchy on right side (implied by role-class inheritance constraint)
  - The template role implies no hierarchy on left side

Diagram:

- DerivedMethod
- Caller ➔ callee ➔ Callee

- TemplateMethod
- Caller ➔ hookObject ➔ Callee ➔ CalleeDescendant

- Template ➔ Hook
Objectifier and Strategy

- **Objectifier has**
  - An additional exclusion constraint on Caller and Callee
  - An aggregation
  - An algorithm role
  - A subclassing constraint (right hierarchy)
  - No template role

- **Strategy is a Objectifier with**
  - Client role
  - Algorithm role
  - Hierarchy on right side
  - No template role
TemplateClass

- TemplateClass is a Objectifier with
  - An additional template/hook role model
  - Template role implies no hierarchy on left side
  - Hook role implies inheritance hierarchy on right side
  - *No client or algorithm role*, otherwise like Strategy
DimensionalClassHierarchies

- DimensionalClassHierarchies is a TemplateClass
  - *Without template-hook constraint*
  - *With left hierarchy constraint*
Bridge

- Bridge is a Dimensional Hierarchies with
  - An additional abstraction/implementation role model
  - *No template/hook role*
Creational Patterns

- Add more roles with semantics about creation
- E.g., FactoryMethod is a TemplateMethod with a creational role model
Remember: Relation TemplateMethod, TemplateClass, Strategy, Observer

More specific patterns (with more intent, more pragmatics)

- Objectifier
- Strategy
- Bridge

- TemplateMethod
- TemplateClass
- Dimensional ClassHierarchies

Different forces
abstracting
concretizing
concretizing

T&H Metapatterns

Framework Patterns
Prof. Uwe Aßmann, Design Patterns and Frameworks
Composition of Simple Extensibility Patterns
Object Recursion

- The aggregation can be 1:1 or 1:n (1-Recursion, n-Recursion)

```
Client
    ▶
    Handler
        ▶
        1 or +
        childObject(s)

        handleRequest()
        preHandleRequest(Component)
        postHandleRequest(Component)

        for all g in childObject(s)
        g.handleRequest()

Terminator
    ▶
    Recurser
        preHandleRequest()
        for all g in childObject(s)
        g.handleRequest()
        postHandleRequest()```
ObjectRecursion

NodeClient \rightarrow Handler

Recurser \rightarrow Terminator

children \{ 1 \ or * \}

RootClient \rightarrow Root
Composite

- n-ObjectRecursion
- Other role pragmatics, similar pattern
- Perhaps with additional parent relation
Decorator

- 1-ObjectRecursion
- other role pragmatics, similar pattern

Diagram:
- NodeClient → Node
- Node → Decorator
- Decorator → Decorated
- RootClient → RootOfList
- 1
- hidden
Chain of Responsibility

- No real ObjectRecursion
Remember: Relations Extensibility Patterns

Specific Patterns

- Decorator
- Composite
- Proxy
- Visitor
- Bridge
- n-Bridge
- Observer

Abstracting

- Dimensional ClassHierarchies
- Recursive T&H Pattern
- Framework Patterns

Still something to discover...
Consequences of the Riehle/Gross Law
Zimmer's Classification and the Riehle-Gross Law

- Zimmer's hierarchy notes use relationships between design patterns
  - But actually, he means composition of role models of design patterns
  - but Zimmer could not express it conceptually
Relations between Patterns [Zimmer, PLOP 1]

Creation patterns
- Prototype
  - Builder
- Abstract Factory
  - Factory Method
- Factory Method

Coupling patterns
- Strategy
- Layers
- Observer
- Bridge
- Facade
- Compositum
- Proxy
- Adapter
- Decorator
- Mediator

Control flow patterns
- Interpreter
- Visitor
- Chain Of Responsibility
- Command
- Iterator
- Memento

Data patterns
- Singleton
- Proxy
- Flyweight

Basic patterns
- Template Method
- Objectifier
- Command
- Flyweight
Consequence for Pattern-Based Design

► When knowing about different role models, the fine semantic differences between several patterns can be expressed syntactically
► This makes designs more explicit, precise, and formal
► You will know when you need a Strategy, TemplateClass, or DimensionalClassHierarchy
Consequence for Pattern Mining

► When you identify a pattern in the product of your company,
  ▪ Try to define a role model
  ▪ Split the role model into those that you know already
  ▪ I.e., decompose the complex pattern in well-known ones

► Advantage:
  ▪ You know how to implement the well-known patterns
  ▪ You can check whether an implementation of the composite, new pattern is correct
  ▪ If all component patterns are implemented correctly, i.e., conform to their role models.
Be Aware: These Role Models Are Not Stable

- Role models provide freedom; so there may be several ones for one pattern
- It would be good to do some PhD theses on those, to make them stable and put them up in a good catalogue
Relation of Role Modelling to Other Software Engineering Technologies
Hyperslice Composition and Role Mapping

- Hyperslices (views) are essentially the same concept as roles
  - But work also on other abstractions than classes and feature sets
  - Hyperslices can be defined on statements and statement blocks
- Role models are more unstructured since they do not prerequisite slices, dimensions, or layers
- Hyperslice composition is similar to role mapping
  - Is guided by a composition that merges views (roles)
  - Hyperslices are independent (no constraints between hyperslices)
- Role models implement aspects
  - Because the roles are related by role constraints
- More in “Component-based Software Engineering”
Roles vs Facets

- A facet is concerned always with *one* logical object
  - A facet classification is a *product lattice*
- Role models may crosscut many objects
  - They are concerned with collaboration of at least 2 objects
  - Hence, a facet is like a role of one object, but from *n* facet dimensions.
  - A class can have arbitrarily many roles, but only *n* facets
- Roles may be played for some time; facets last over the entire lifetime of the object
Role Types Formally

- A semantically rigid type expresses:
  - If an object that has the type, it cannot stop being of the type without losing its identity
  - Semantically rigid types are tied to the identity of objects

- Example:
  - A Book is semantically rigid. A Reader can stop reading, but a Book stays a Book

- A semantically rigid type is tied to a class invariant (holds for all objects at all times)
Role Types Formally

- A *semantically non-rigid type* is a dynamic type that is indicating a state of the object.

- A *founded type* is a type if an object of the type is always in collaboration (association) with another object.
  - Example: Reader is a founded type because for being a reader, one has to have a book.

- A *role type* is a founded and semantically non-rigid type.
  - Role types are in collaboration and if the object does no longer play the role type, it does not give up identity.

- *Natural types* are non-founded and semantically rigid.
  - Book is a natural type.
  - A natural type is *independent* of a relationship.
  - The objects cannot leave it.
Effects of Role-Based Design Patterns on Frameworks and Applications
Effect of Role Models

- Role modelling allows for \textit{scaling of delegation}
  - By default, all roles are overlaid by their class
  - But some can stay separate
  - Layered frameworks split all roles off to role objects
Role Models and Facet/Layered Frameworks

- An n-Bridge framework maintains roles (role models) in every facet (because a facet model is based on a class-role model)
- Similar for chain-Bridges and layered frameworks

Diagram:

- Core Layer: Abstraction Framework
- First layer
- Second layer
- Third layer
- Reuse 0
- Reuse 0 & 1
- Reuse 0-2
- Reuse 0-3
Merging Layers of Facet/Layered Frameworks

- If the layers are seen as role models, it can be chosen to merge the layers, i.e., the role models
- Here: merge second and third layer into one physical implementation layer
- No reuse for layer 2 possible

Diagram:

- Core Layer: Abstraction Framework
- Reuse 0
- Reuse 0&1
- Reuse 0-3

First layer

Second layer

Third layer
Merging Layers of Layered Frameworks

► When two layers are merged, the variability of a framework sinks
► But its applications are more efficient:
  ▪ Less delegations (less bridges)
  ▪ Less allocations (less physical objects)
  ▪ Less runtime flexibility (less dynamic variation)
The roles of MVC can be ordered in a n-Bridge framework.

- First layer: Views
- Second layer: Controller
- Third layer: Model

Core Layer: Application

Reuse 0
Reuse 0&1
Reuse 0-2
Reuse 0-3
MVC as Optimized n-Bridge Framework

- Model and Controller layer can be merged
- Less variability, but also less runtime objects
Optimization of Design Patterns with Role Models
Aßmann's Law of Optimization for Design Patterns :-) 

Whenever you need a variant of a design pattern that is more efficient, investigate its role model and try to merge the classes of the roles.

- Effect:
  - Less variability
  - Less runtime objects
  - Less delegations
Original Role-Class Model of MVC

Prof. Uwe Aßmann, Design Patterns and Frameworks
Optimized Role-Class Model of MVC

Model
- StrategyClient (Strategy)
- Subject (Observer)
- DirectorClient (Composite)

Controller
- Strategy (Strategy)
- ClerkClient (Composite)

View
- Component (Composite)
- Observer (Observer)
- ComposedView (Composite)
- LeafView
- RootView (Composite)

Model\'New

View\'New

Subject (Observer)
Optimized Role-Class Model of MVC

- The optimized model merges all roles into two classes
  - No strategy variation
  - No composite views
- Only 2 instead of 3+n objects at runtime
  - Faster construction
  - Essence of the pattern, the Observer, is still maintained
- However, restricted variability
Super-Optimized Role-Class Model of MVC

ClassBeingNoLongerAnMVC

Model
- StrategyClient (Strategy)
- Caller (Call)

Controller
- Strategy (Strategy)
- ClerkClient (Composite)
- Component (Composite)
- Callee (Call)

View
- Composed View
  - Composed (Composite)
- LeafView
  - Leaf Composite

Root View
- Root (Composite)
In this design, the ClassBeingNoLongerAnMVC merges all roles:
- It should be a superclass of all contained classes.
The Observer pattern is exchanged to a standard call.
- No variability anymore.
- But only one runtime object!
Summary

- Roles are important for design patterns
  - If a design pattern occurs in an application, some class of the application plays the role of a class in the pattern
  - Roles are dynamic classes: they change over time
- Role-based modelling is more general and finer-grained than class-based modelling
- Role mapping is the process of allocating roles to concrete implementation classes
- Hence, role mapping decides how the classes of the design pattern are allocated to implementation classes (and this can be quite different)
- Composite design patterns are based on role model composition
- Layered frameworks and design patterns can be optimized by role merging
The End