



⊕ Software Security: Design and code

Yves Le Traon

Tejeddine Mouelhi



➔ Overview

- **Security: definitions and “big picture”**
- **Example of security weakness: SQLIA**
- **Security in the development process**
- **Security requirements**
- **Security analysis and design**
- **Security test and validation**



➔ Security : general definition

- **Protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction.**
- **CIA**
 - **Confidentiality**
 - accessed, used, copied, or disclosed by persons who have been authorized to access, use, copy, or disclose the information
 - **Integrity**
 - data can not be created, changed, or deleted without authorization
 - **Availability (and correctness) of**
 - the information and the security controls (opposite of availability is denial of service - DOS)
- **Confidentiality, possession or control, integrity, authenticity, availability, and utility.**
- **Ex of approach for confidentiality: Encryption/cryptography**

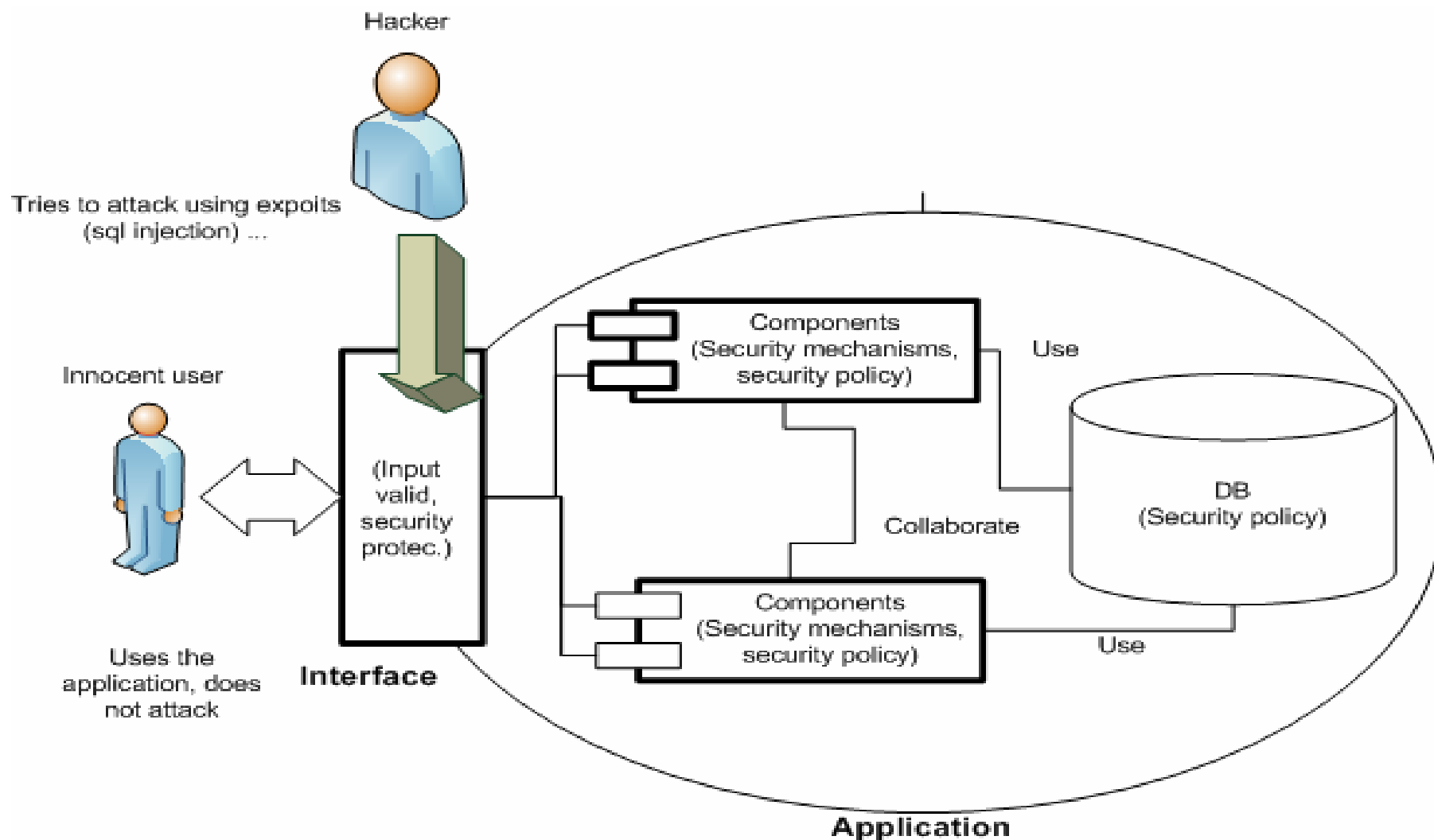


➔ Vocabulary

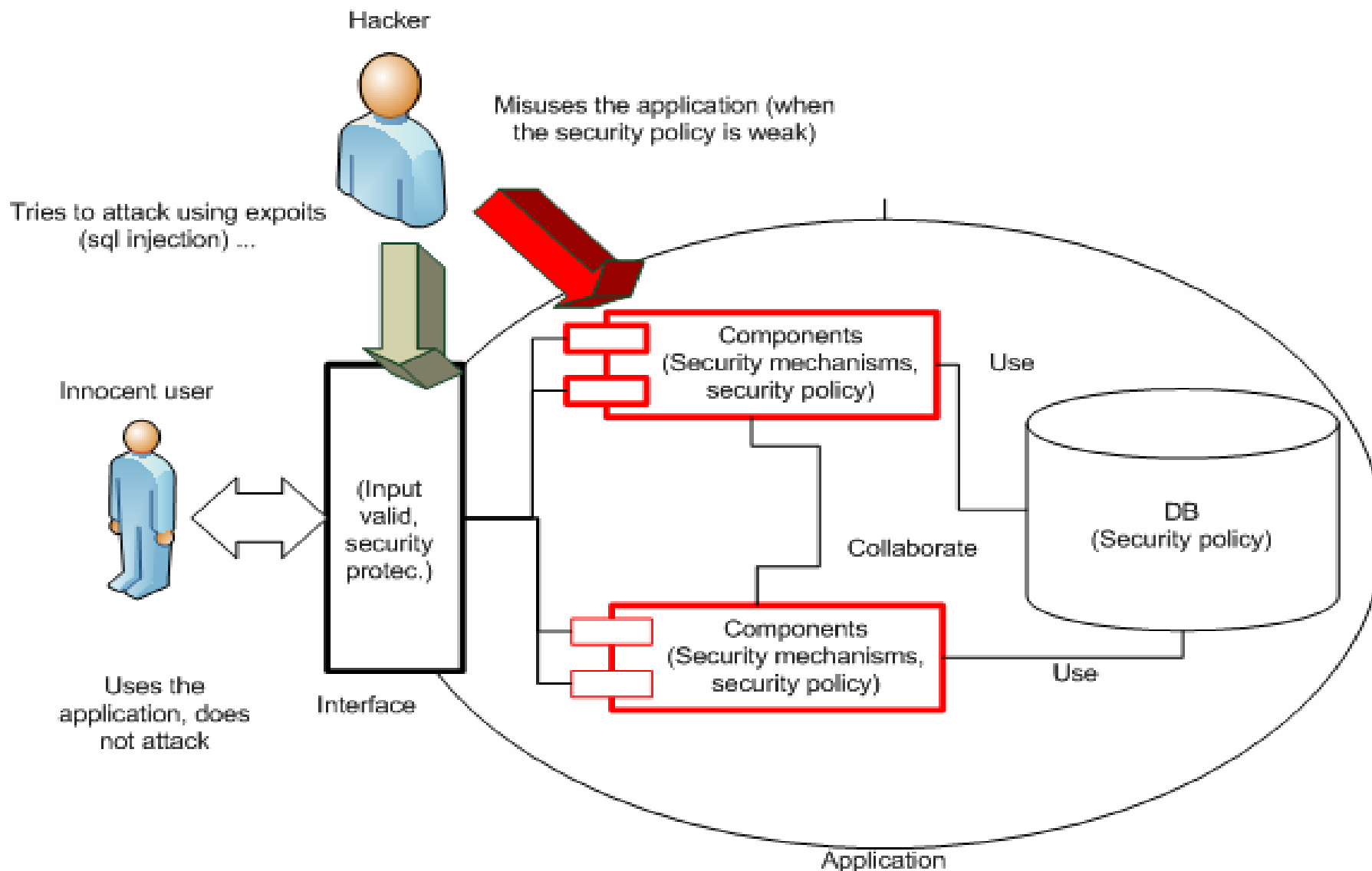
➤ Risk management

- **Risk** is the likelihood that something bad will happen that causes harm to an informational asset (or the loss of the asset).
 - A **vulnerability** is a weakness that could be used to endanger or cause harm to an informational asset.
 - A **threat** is anything (man made or act of nature) that has the potential to cause harm.
- **A security mechanism is the implementation of a security requirement (e.g. access control rule)**

⊕ Securing a web application



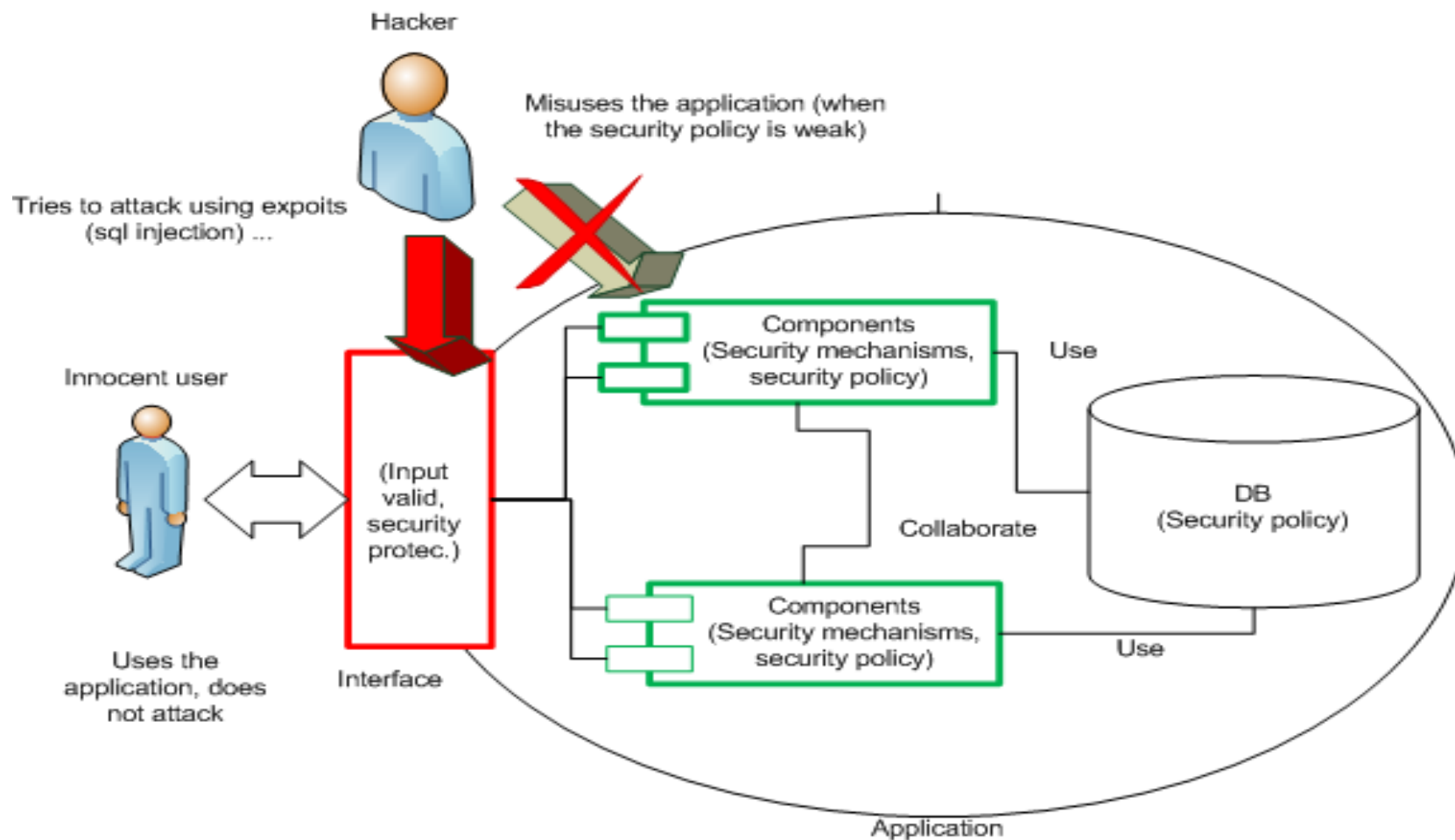
⊕ Securing a web application



⊕ Les couches de sécurité

Application	Mécanismes de sécurisation du code, la logique métier ...
Système	Antivirus, anti-spyware, anti-malware ...
Réseau	Cryptographie, pare feu, protocoles sécurisés, IDS ...

⊕ Security weakness : an example (SQLIA)





⊕ L'injection de code SQL (SQLIA)

➤ Caractéristiques

- Cause : Requêtes dynamiques utilisant les entrées de l'utilisateur non filtrées
- Code SQL injecté par l'utilisateur et compilé dans la requêtes SQL.
- Conséquences: modification, détournement des données voire suppression.

➤ Exemple:

- La requête :
 - `select * from users where login = ' + varLogin + ';`
- L'utilisateur entre la valeur:
 - `' or 1=1`
- La requête devient une tautologie:
 - `select * from users where login = " or 1=1;`

⊕ SQL Injection countermeasure

➤ Input validation methods

- Message data to get valid data:
 - Add a '\' before all malicious characters
 - The character will not be interpreted ('-' becomes '/--')
 - But, the list of malicious character is not final (new unknown character may be used such as '#' which has been introduced to deal with dates)
- Reject illegal characters: delete all malicious characters (like ' or --)
 - Same problems with the previous approach
- Accept only authorized characters
 - Will always work.
 - But we will need to refuse to save in the DB names like (o'connor) and no words containing – or \
- => Best method is to combine the last two approaches.

⊕ SQLIA : Contre-mesures avancées

➤ Plusieurs techniques avancées utilisant des mécanismes de sécurité indépendants de l'application

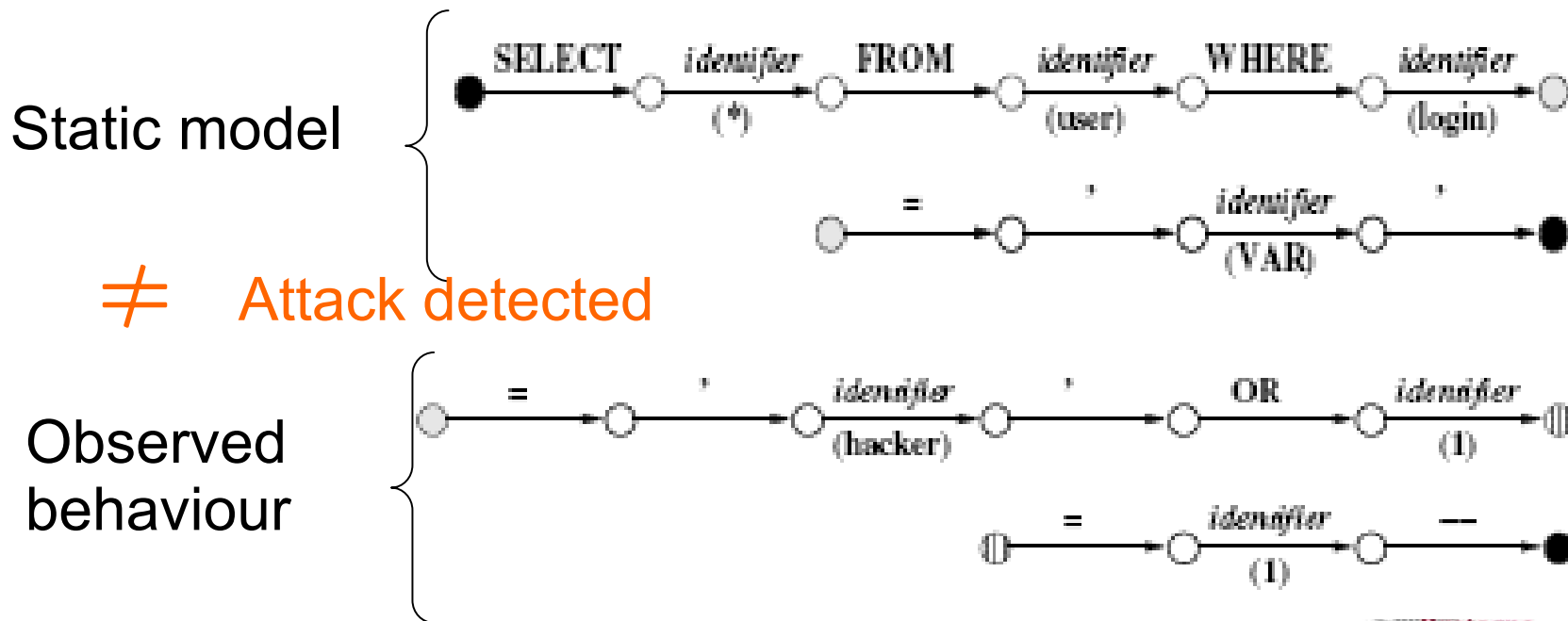
- Static vs. runtime analysis
- Machine learning
- SQL Rand



Advanced approaches

combining static and runtime analysis.

- SQL Finite State Machine (SQL-FSM) is created for all queries statically by scanning the code.
- These SQL-FSM are compared to dynamic query at the execution
- If the model is not the same, then the attack is detected.

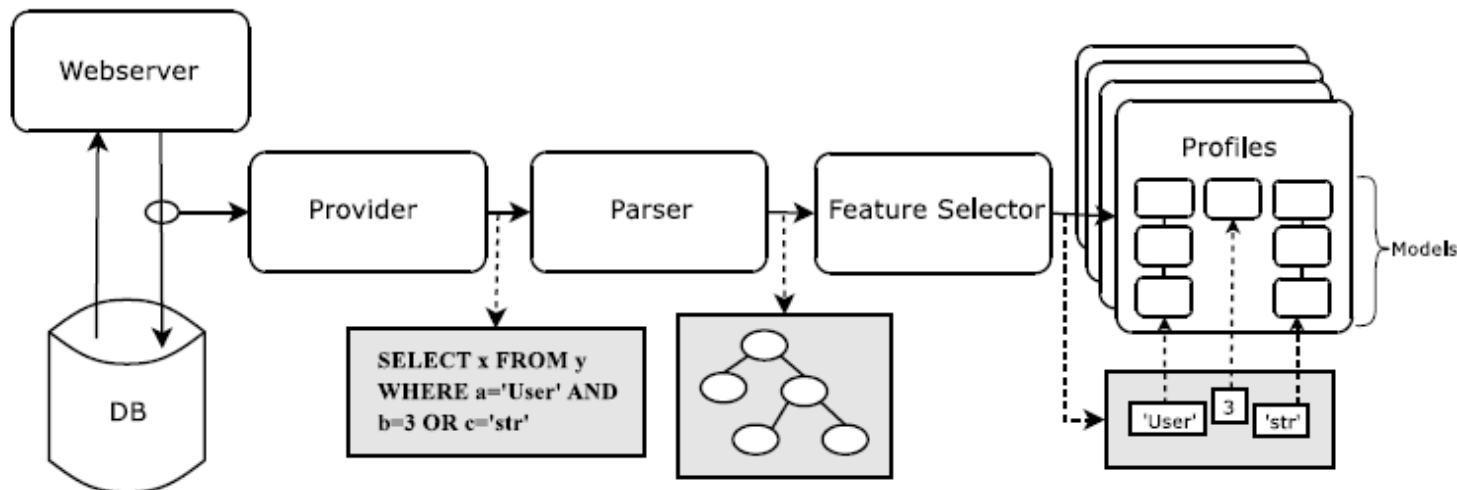


⊕ Advanced approaches

13 -

• Machine learning based approach

- based on the idea that an application always performs the same types and models of SQL queries
- if a new kind of query is performed then this is an indicator of a possible SQLIA.
- This approach works as follows:
 - During a training period the security mechanism learns all the models of queries that can be performed by the application.
 - The security mechanism monitors checks every query performed by the application to see if it is linked with an already known model. If it is not then an alert is raised.





⊕ SQLIA : Contre-mesures avancées

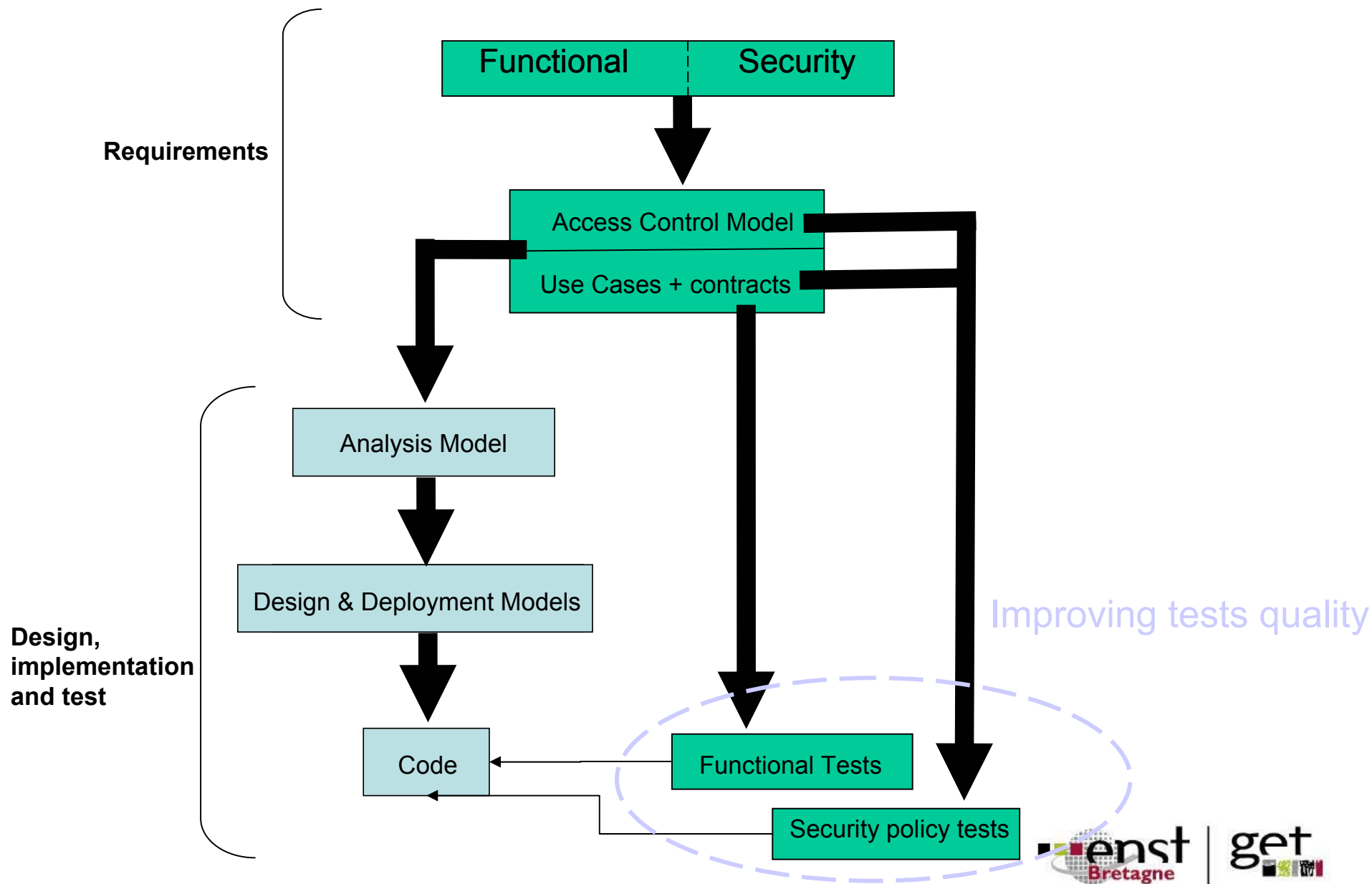
- Plusieurs techniques avancées utilisant des mécanismes de sécurité indépendants de l'application:
 - Exemple : Basé sur la randomisation des requêtes (une clé) (SQLRand)
- Un exemple SQLRand:
 - La requête :
 - `select * from users where login = ' + varLogin + ';`
 - Randomization de la requête (la clé est 123)
 - `select123 * from123 users where123 login = ' + varLogin + ';`
 - En injection `' or 1=1` :
 - `select123 * from123 users where123 login = " or 1=1; (=> or123)`
 - La requête erronée => échec de l'attaque

⊕ **Morality and objectives**

- **Analyse security requirements**
- **Expert knowledge useful**
- **Design for security**
- **Test security**

- **Criticality of the development process**

Security in the development process





⊕ Security requirements: a library management system

- offer services to manage books in a public library
- books can be borrowed and returned on working days. When the library is closed, users can not borrow books. When a book is already borrowed, a user can make a reservation for this book. When the book is available, the user can borrow it.
- managed by an administrator (create, modify and remove accounts for new users).
- A secretary who can order books, add them in the LMS when they are delivered. The secretary can also fix the damaged books (maintenance days)
- The director of the library has the same accesses than the secretary and he can also consult the accounts of the employees
- The administrator and the secretary can consult all accounts of users.
- All users can consult the list of books in the library
- Three types of users: public users who can borrow 5 books for 3 weeks, students who can borrow 10 books for 3 weeks and teachers who can borrow 10 books for 2 months

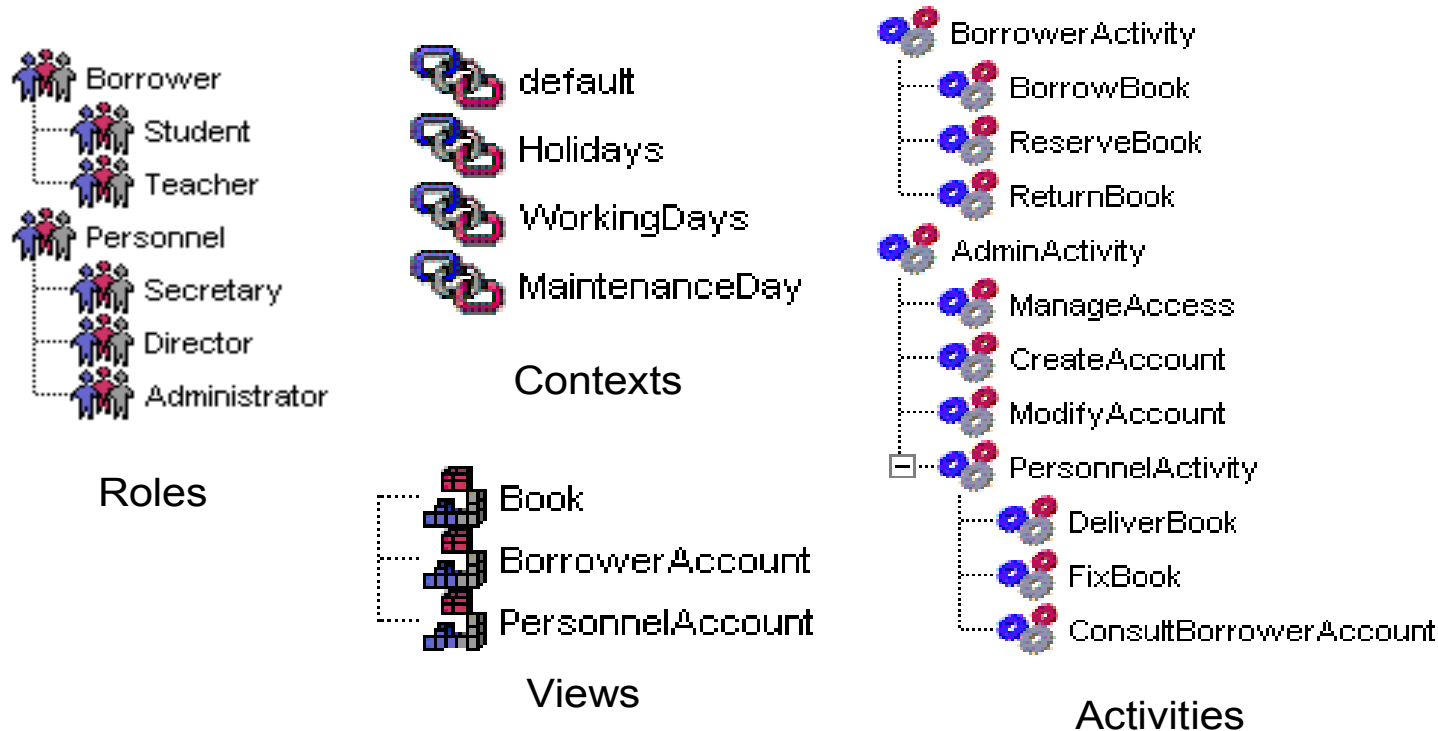
⊕ Security policies: Access control models

- **Informal definition:**
 - Express permissions or prohibitions for people to access any of the resources of the system.
- **May take into account the circumstances or contexts (delegation, temporal or spatial properties)**
 - In the LMS, we must distinguish between working days, holidays and maintenance days.

OrBAC

- **OrBAC : Organization based access control.**
- **Rules of permission, prohibition or obligation**
 - Rule signature : the organization, the role, the activity, the view and the context
 - *Permission(org,role,activity,view,context).*
 - Example:
 - *Permission(Library,Borrower,Borrow,Book,Holidays).*
- **Hierarchies, rules derivation and conflict management.**
- **A tool : MotOrBAC**

OrBAC



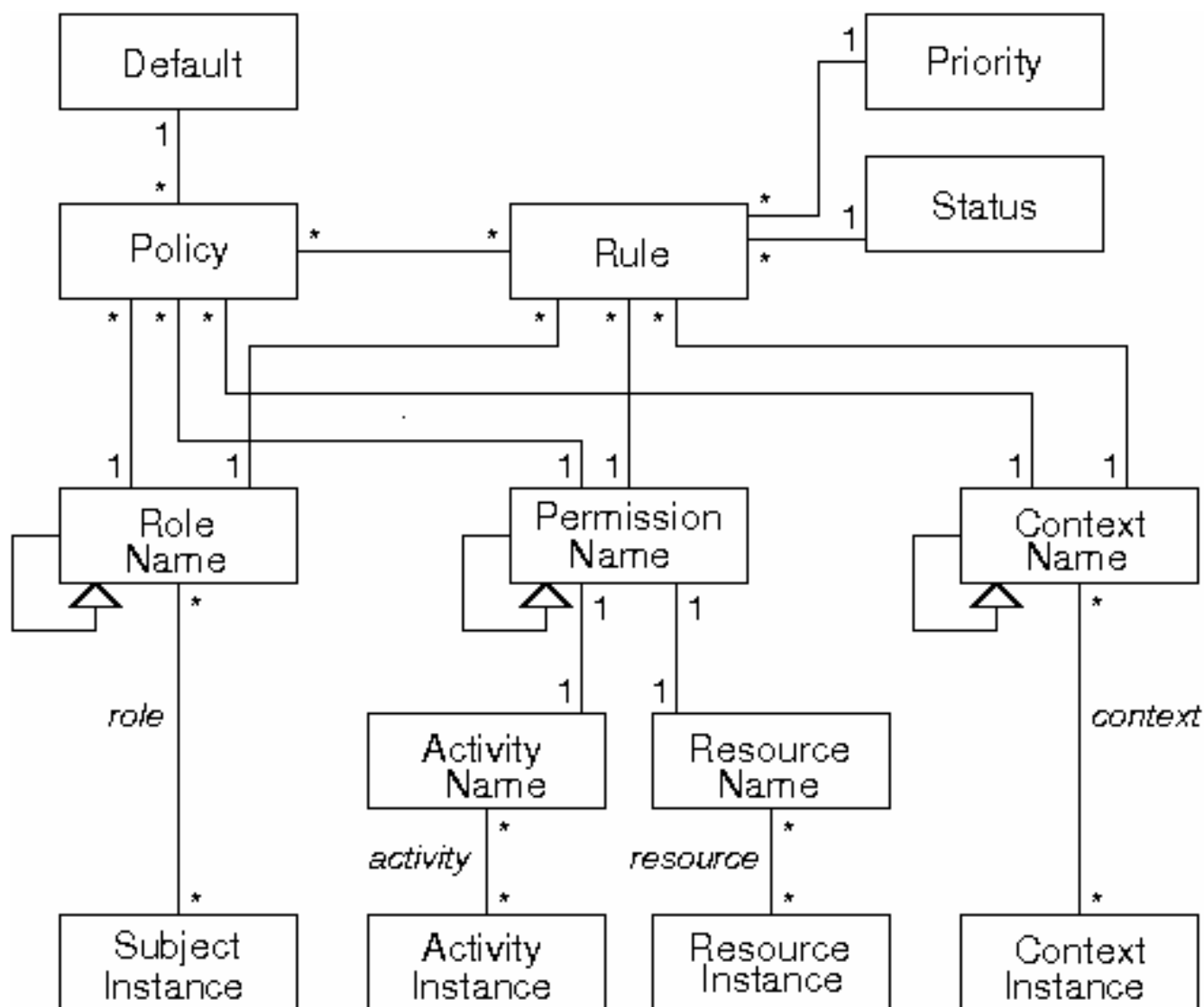
➤ **20 primary rules, 22 derived rules, 35 concrete rules**

➤ **Examples of rules**

➤ *Permission(Library,Administrator, ModifyAccount, BorrowerAccount, WorkingDays).*

➤ *Permission(Library,Personnel, FixBook, Book, MaintenanceDay).*

Security policy – class model



➔ Security policy: conclusion

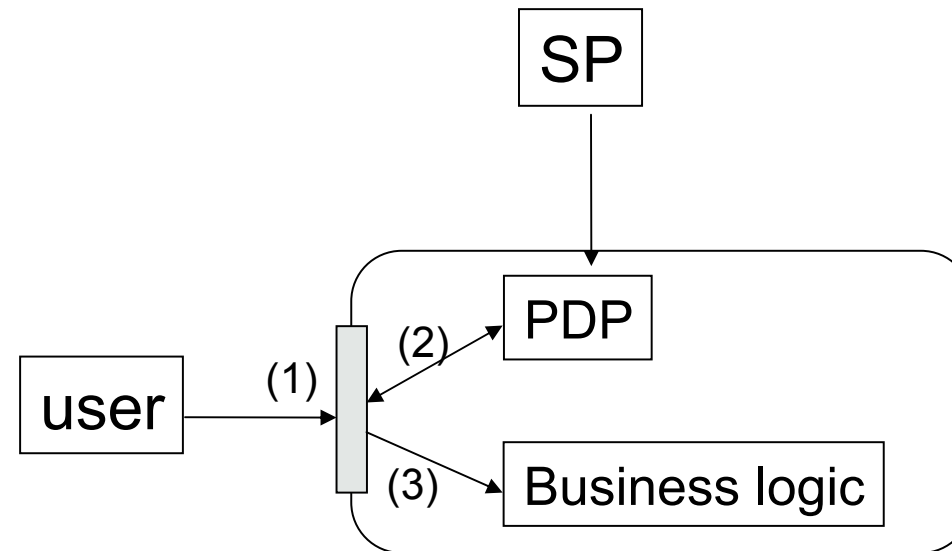
- **Express security requirements in terms of access control policy**

- **Requirements are not the final product !**

- **Key issues: how to**
 - design a secure architecture ?
 - implement security requirements ?
 - Validate that the final system is consistent with requirements ?

⊕ Design and architecture

➤ Conceptual representation in three parts



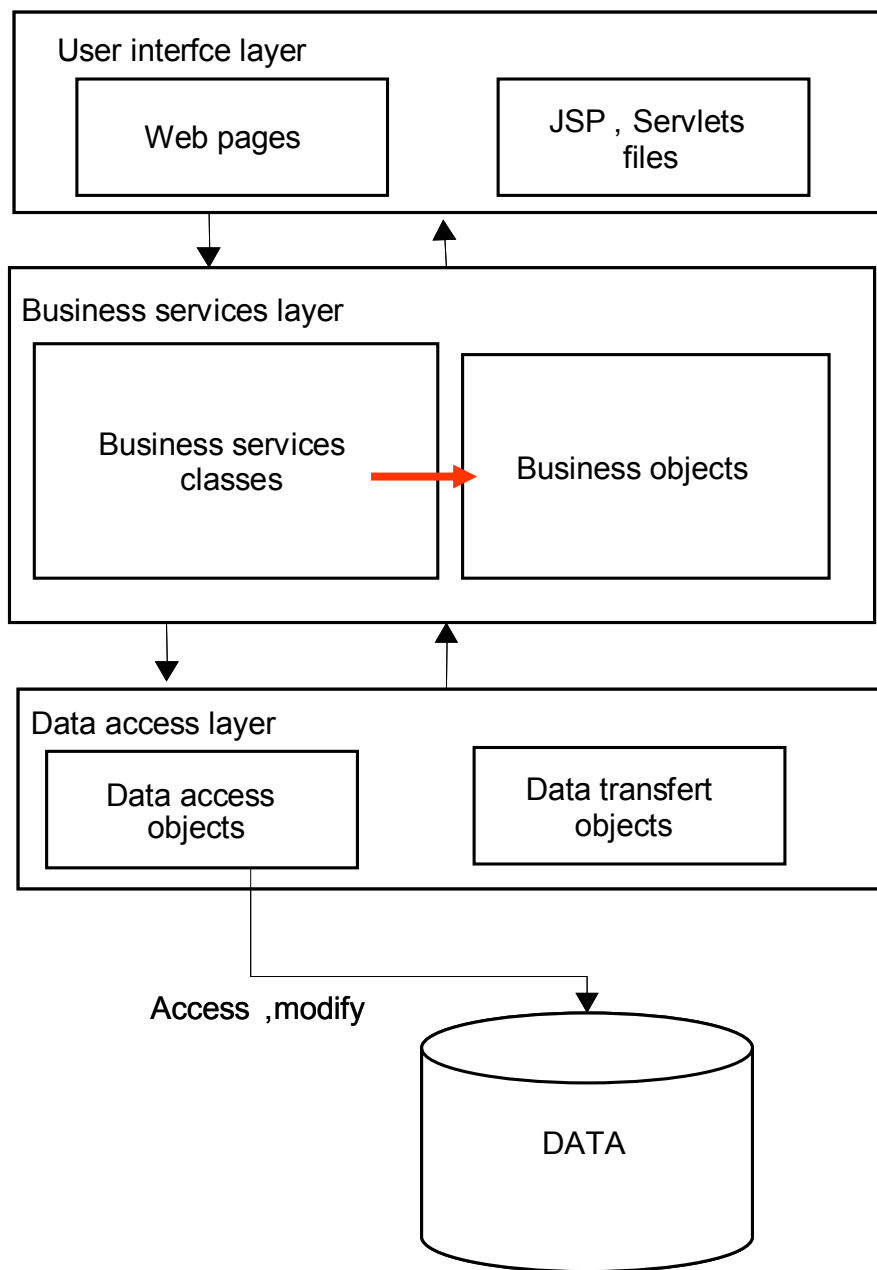
SP: Security Policy

PDP: Policy Decision Point

➔ Ideal architecture for a secure system

- **Low coupling between security mechanisms and business objects**
- **Traceability between security requirements and security mechanisms**
- **Separate security components:**
 - Control and correction
 - More evolvable
 - BUT : the business logic design must be flexible

Implementation: the LMS architecture

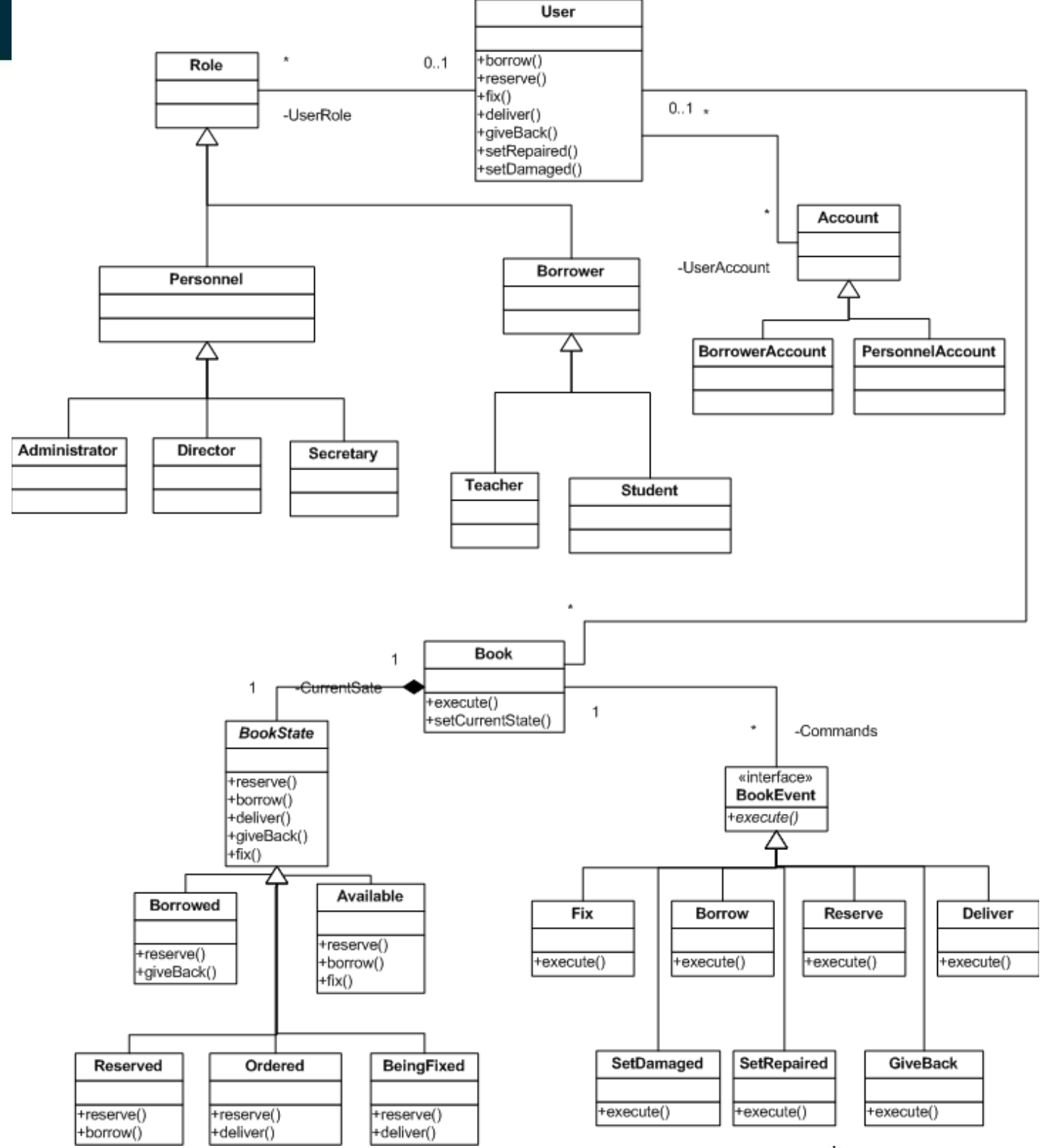


Java
 3200 lines of code
 62 classes
 335 methods.



Functional design

business objects



⊕ Business services

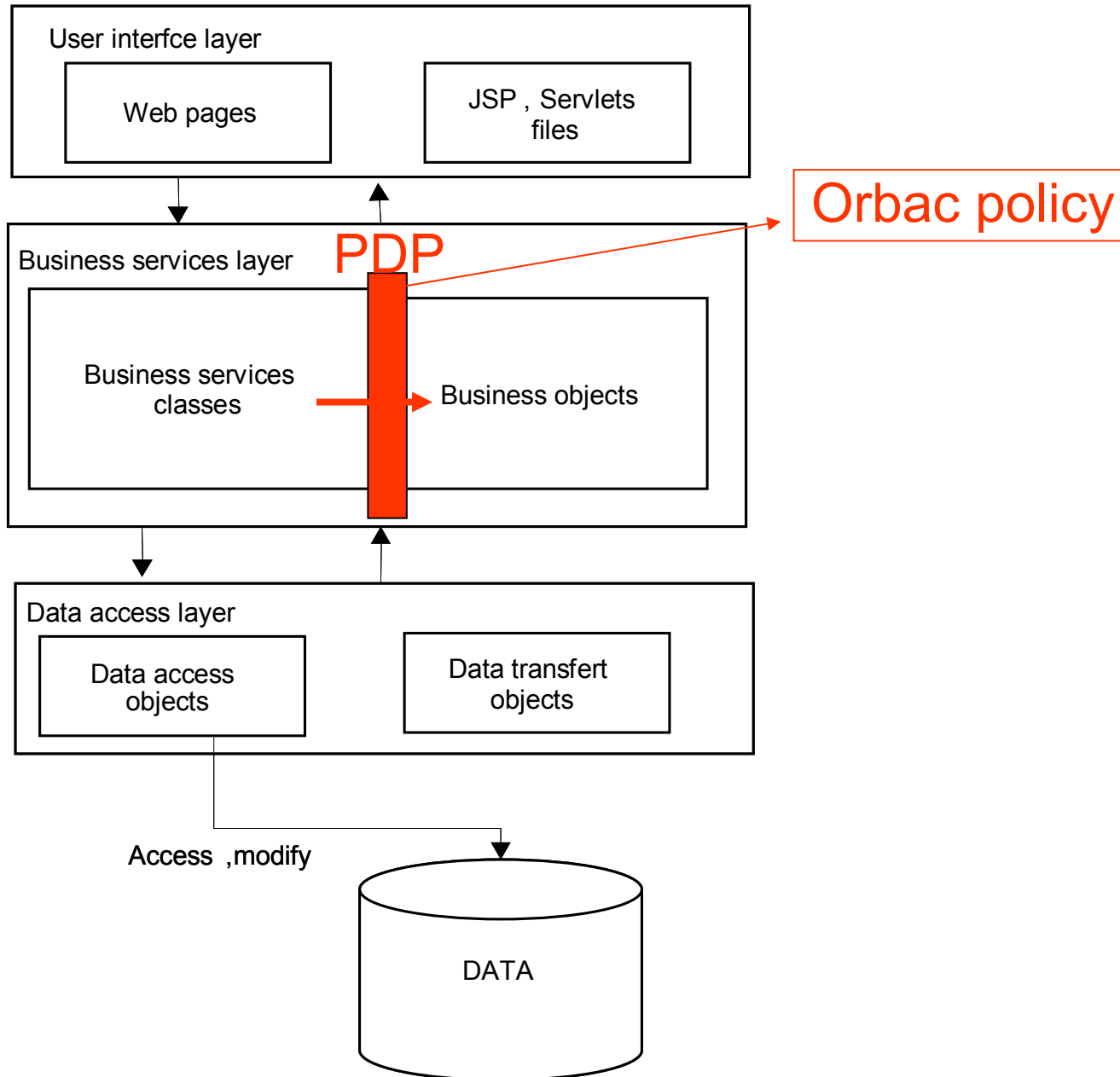
➤ **bookService**

- provides : reserveBook, borrowBook, returnBook.

➤ **Interface et bout de code**

```
public void borrowBook(User user, Book book) {  
  
    // call to business objects  
    // borrow the book for the user  
    book.execute(Book.BORROW, user);  
  
    // remove the book from reserved  
    user.getReservations().remove(book);  
  
    // call the dao class to update the book in the DB  
    // update book in the DB  
    bookDAO.insertBorrow(userDTO, bookDTO);  
}
```

Implementation: the PDP



⊕ PDP implementation

➤ check the security policy when a business service is called

➤ **bookService**

· provides : reserveBook, borrowBook, returnBook.

➤ **Code**

```
public void borrowBook(User user, Book book) throws SecurityPolicyViolationException {
```

```
// call to the security service (security code)
```

```
ServiceUtils.checkSecurity(user,  
LibrarySecurityModel.BORROWBOOK_METHOD,  
LibrarySecurityModel.BOOK_VIEW,  
ContextManager.getTemporalContext());
```

```
// call to business objects
```

```
// borrow the book for the user  
book.execute(Book.BORROW, user);
```

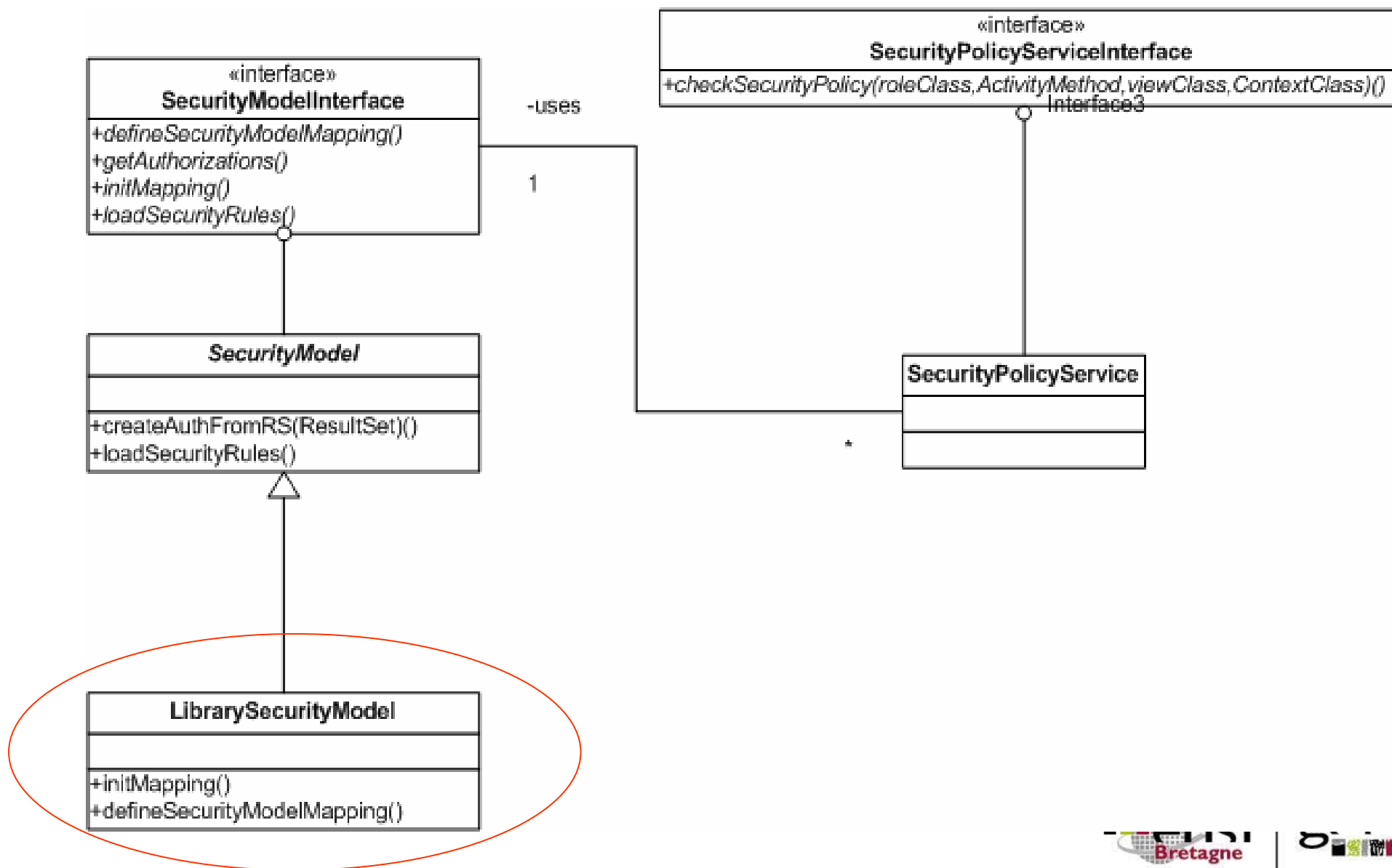
```
// remove the book from reserved  
user.getReservations().remove(book);
```

```
// call the dao class to update the book in the DB
```

```
// update book in the DB  
bookDAO.insertBorrow(userDTO, bookDTO);
```

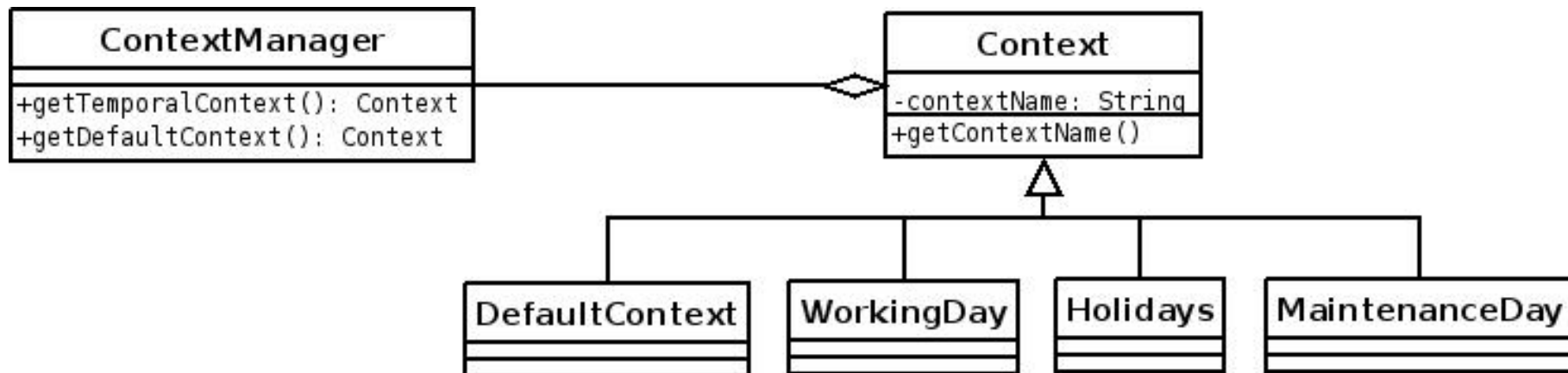
```
}
```

Policy Decision Point



⊕ PDP contexts

- Le contexte est une notion qui n'est pas nécessairement connue du modèle métier (business objects)
- Doit apparaître explicitement dans le PDP

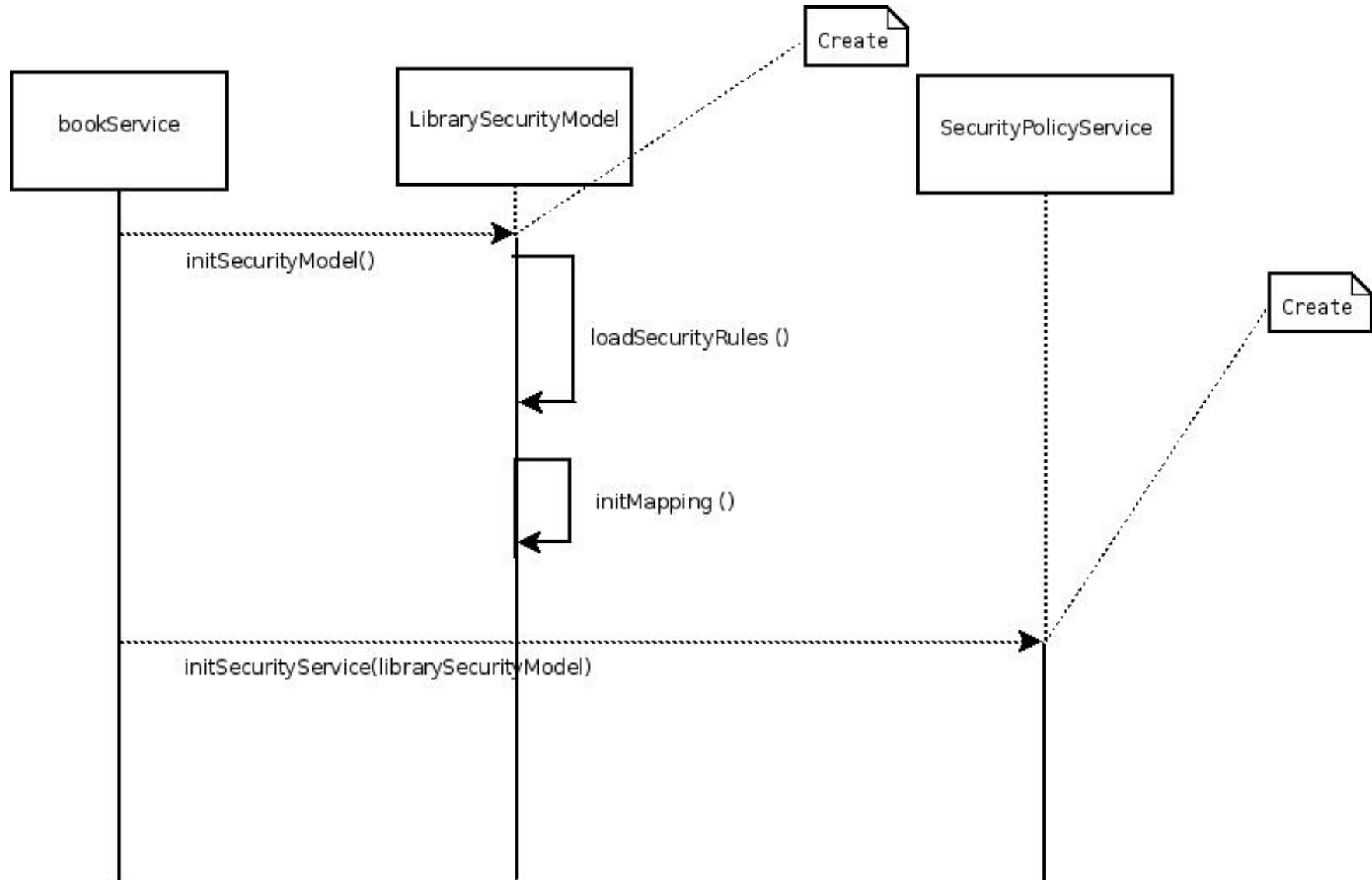




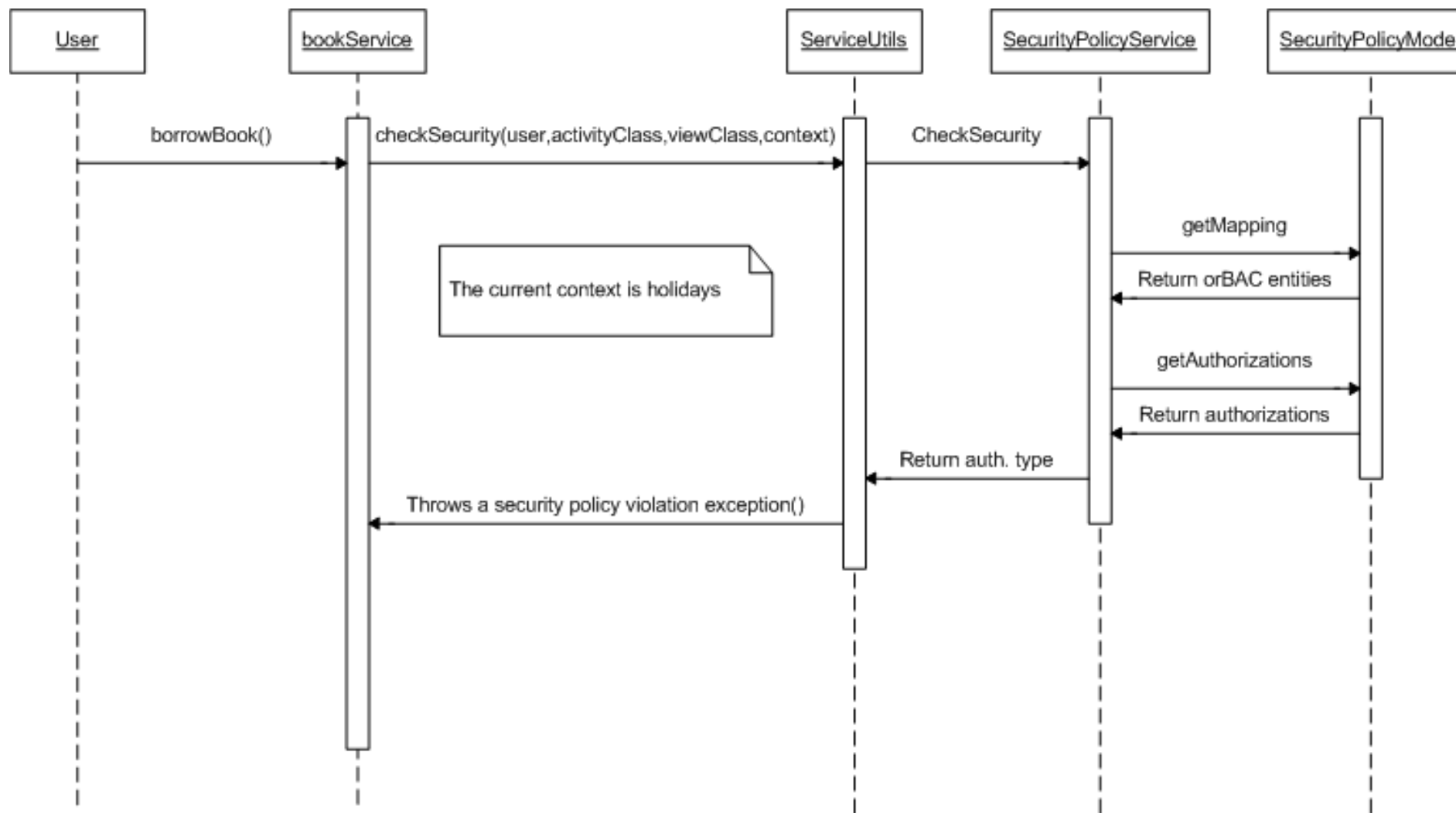
⊕ Mapping table

- **Maps OrBAC concepts to business objects (traceability)**
 - . The library system (Application entity -> OrBAC entity)
 - . Roles
 - . Student class -> Borrower role
 - . Admin class -> Administrator role
 - . Activities:
 - . updateUserAccount method -> Update activity
 - . updatePersonnelAccount method -> Update activity
 - . borrowBook method -> Borrow activity
 - . Views:
 - . Book class -> Book view

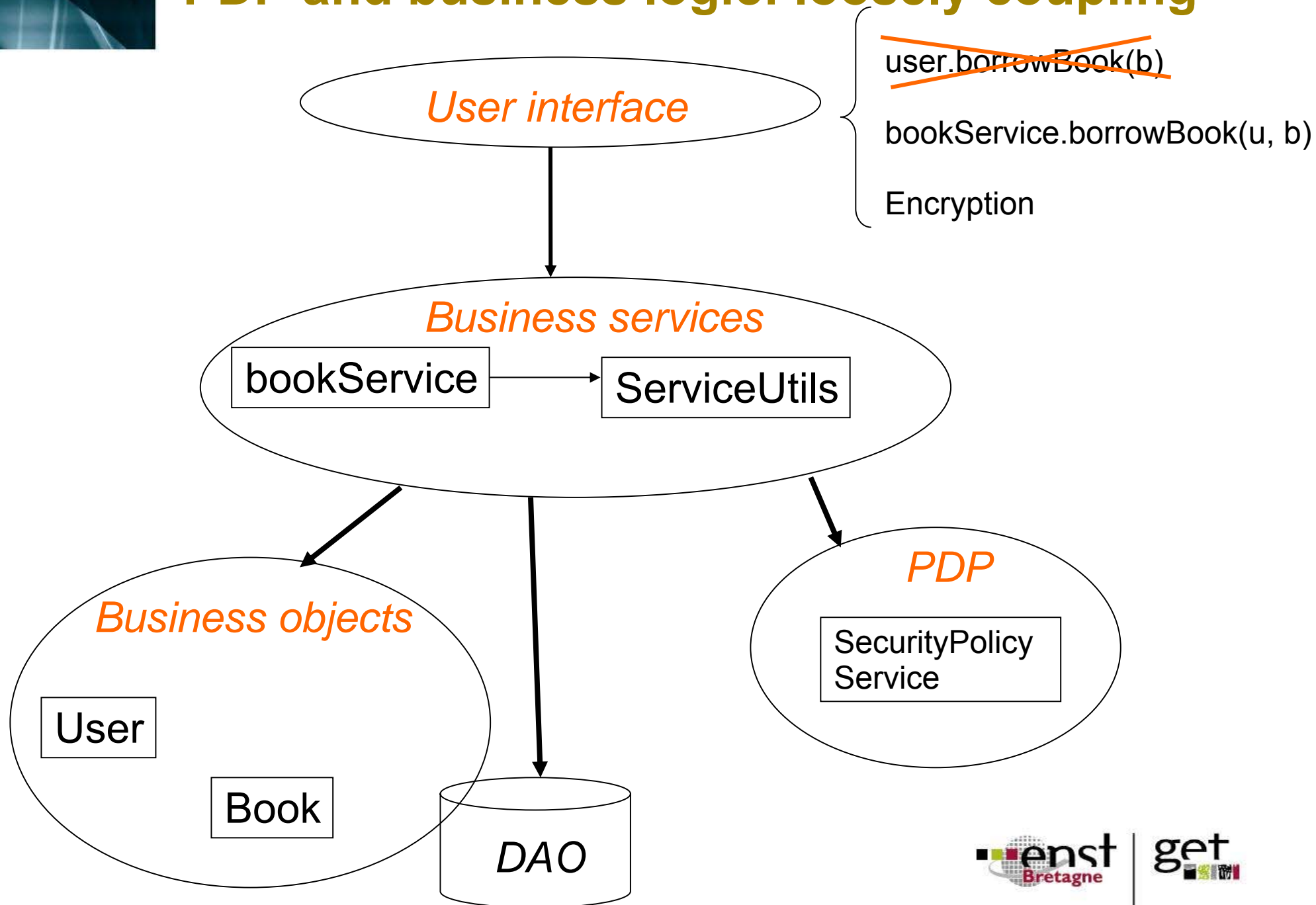
⊕ PDP initialization



⊕ PDP and business logic interactions



⊖ PDP and business logic: loosely coupling

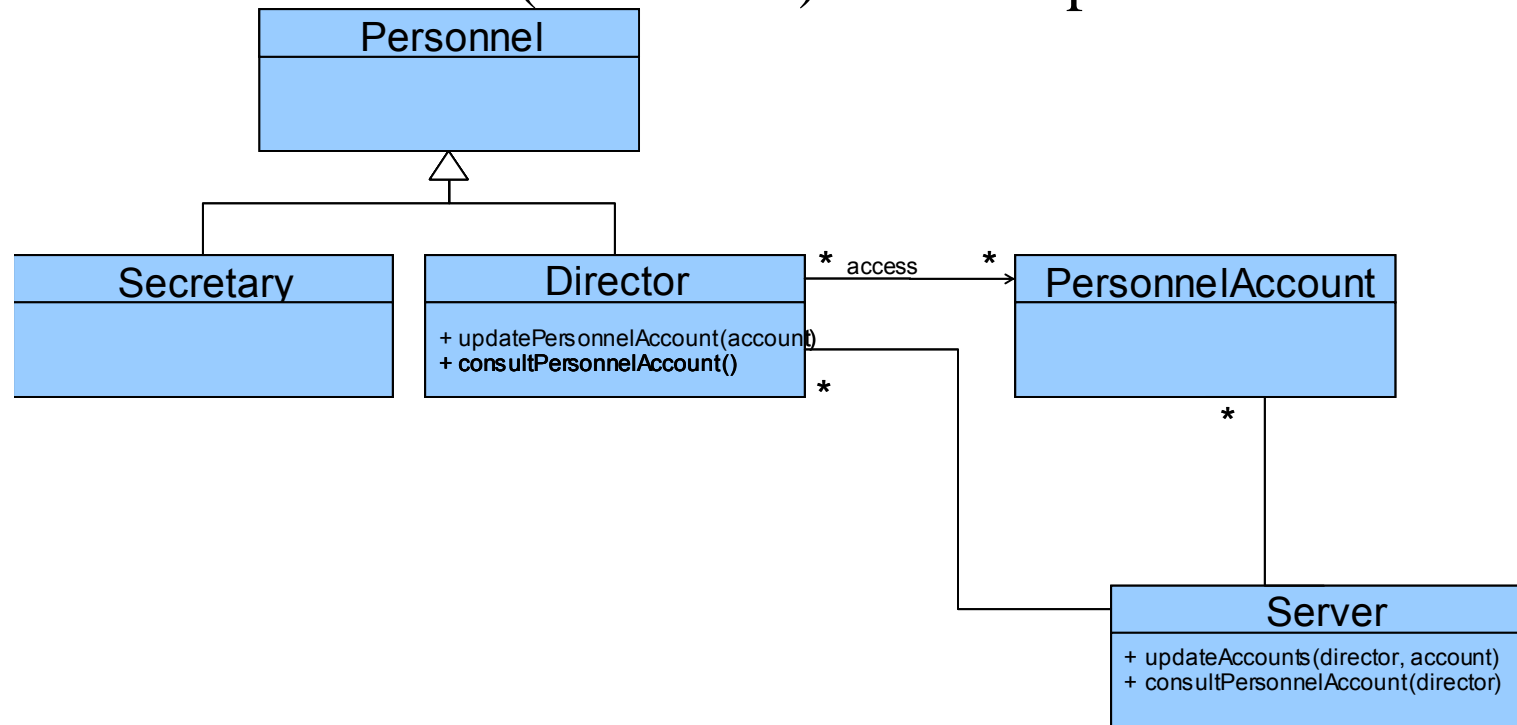


⊕ The problem of legacy systems

- Existing system
- Large scale
- Documentation incomplete
- Access control policy partially known
- Need for evolution/migration/merge
- Problems of design flexibility

➔ Legacy systems: Business design flexibility

- rules may be written in code or implicit in the class model.
- **Example of hard-coded design: adding permission(library, secretary, update, account, default)**
 - By construction, a secretary cannot updatePersonnelAccount
 - The server class (the facade) does not provide this service

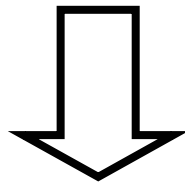
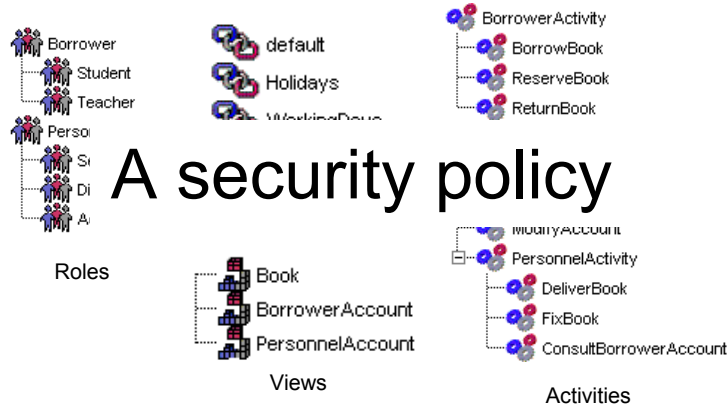


⊕ How to test security

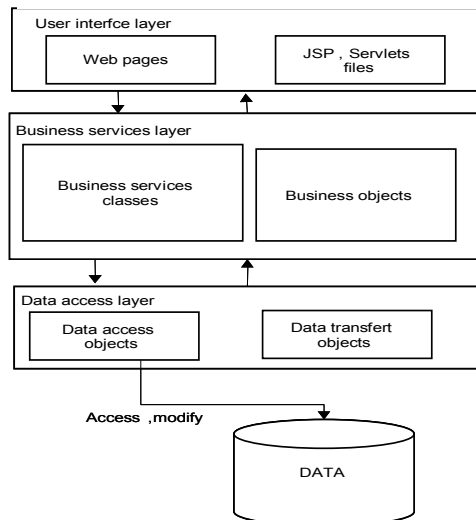
- Explained in a course dedicated to software testing



➔ Security policy testing



Implementation



Test cases



➔ Test case: an example

➤ *Intent*

- permission for a borrower to borrow a book the working days.

➤ *Test sequence*

- create the sequence reaching the context of a working day,
- make a borrower borrow a book.

➤ *Oracle*

- interrogate the security mechanism and check that the permission has been computed and given to the borrower.



➔ Test criteria

➤ **Functional test cases**

- system tests, generated based on the use cases.

➤ **CR1**


- a test case for each primary access control rule of the security model.

➤ **CR2**

- a test case is generated for each primary and secondary rule of the security model, except the generic rules.

➤ **Advanced SP test cases**

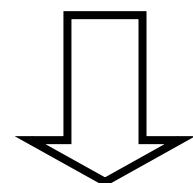
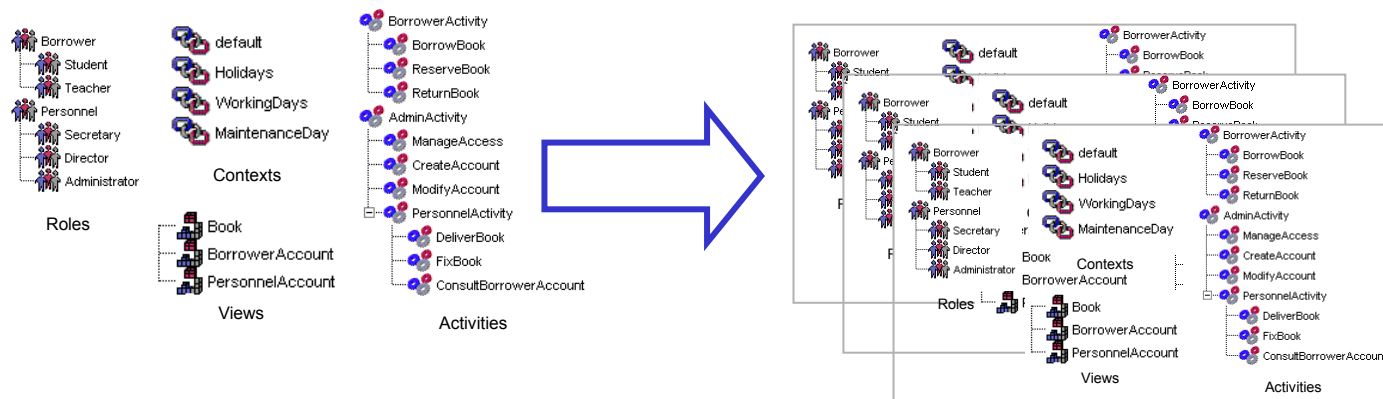
- exercise the default/non specified aspects of the security policy.



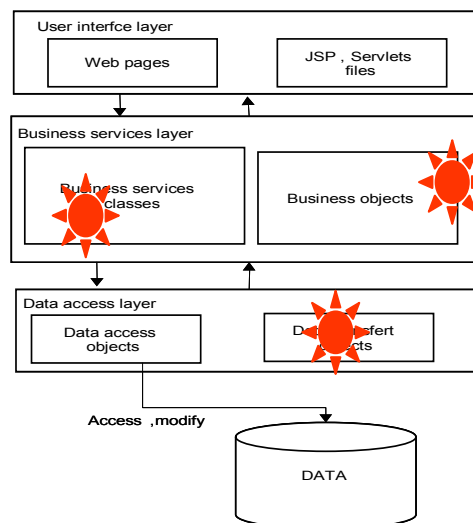
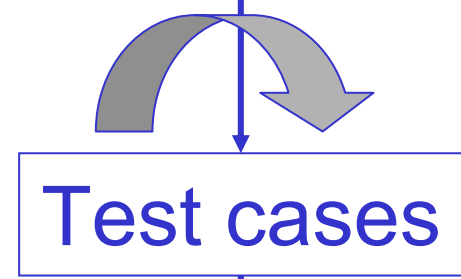
➔ Reusing functional test cases for security purpose ?

- Reusing the test input sequence
- Modifying the intent and the oracle
- *Incremental strategy: It denotes the strategy for producing security test cases which reuse existing test cases.*
- *(in the other case, the strategies are independent)*

➔ Security policy mutation analysis



Mutants implementation

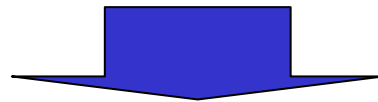


➔ Mutation operators

➤ Simulating access control flaws

- **Permission and prohibition** – Rule's type errors (PRP-PPR)

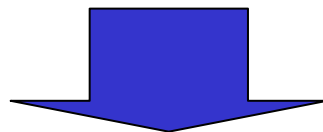
Permission(Library, Administrator, ModifyAccount, BorrowerAccount, WorkingDays)



Prohibition(Library, Administrator, ModifyAccount, BorrowerAccount, WorkingDays)

- **Role and context** – Parameter errors (RRD-CRD).

Permission(Library, Administrator, ModifyAccount, BorrowerAccount, WorkingDays)

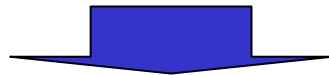


*Permission(Library, **Teacher**, ModifyAccount, BorrowerAccount, WorkingDays)*

➔ Mutation operators

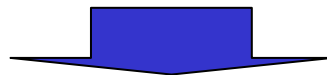
- **Hierarchy errors** on roles and activities (RPD-ADP)

Permission(Library, Student, BorrowerActivity, Book, WorkingDay)



*Permission(Library, Student, **Reserve**, Book, WorkingDay)*

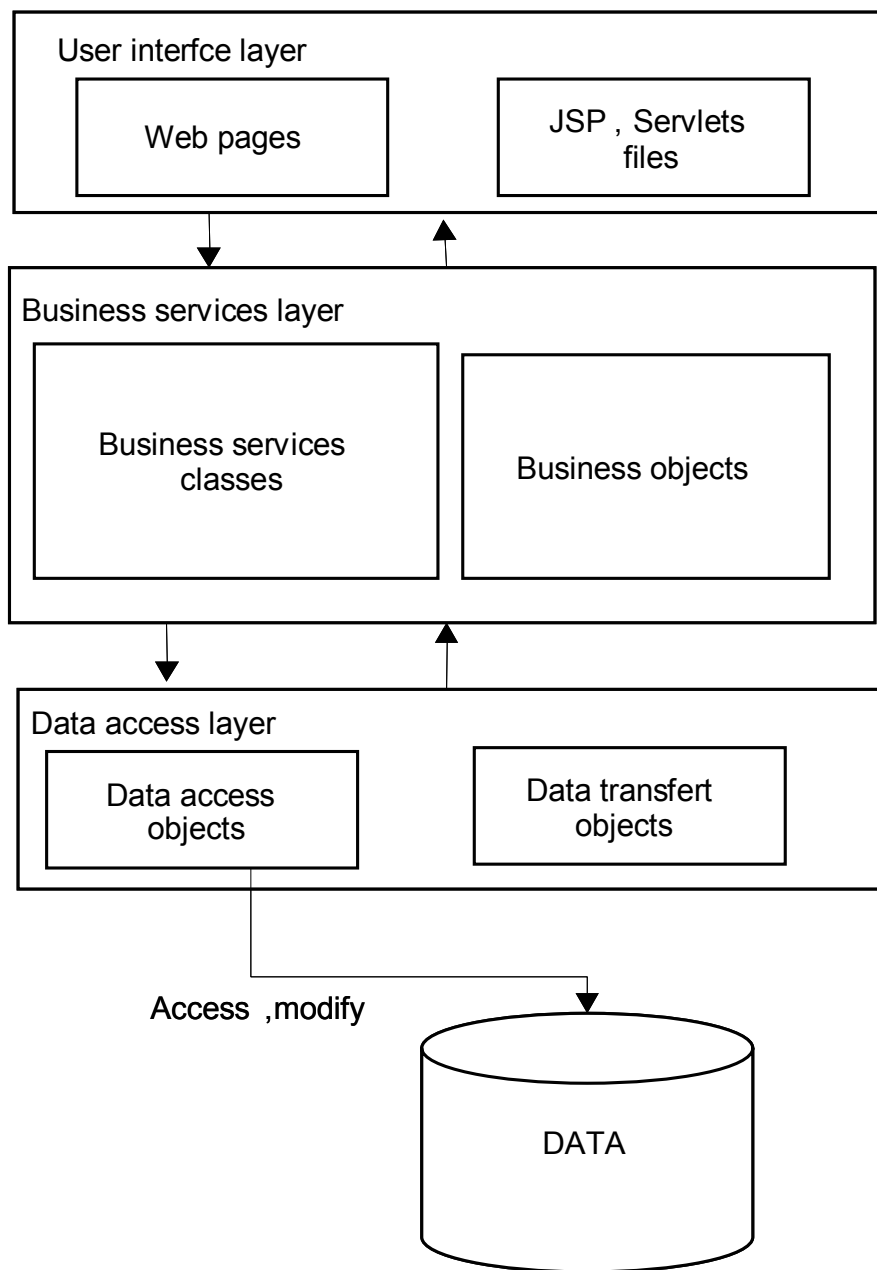
- **Rule addition** for checking tests robustness (ANR)



*Prohibition(Library, **Personnel**, reserve, Book, WorkingDay)*

- **Property** : A mutated rule has a higher priority than the other rules

➔ The LMS architecture



Java
 3200 lines of code
 62 classes
 335 methods.

➔ Generated operators

Operator category		Operator name	Number of mutants
Basic Mutation operators	Rule type changing operator	PPR	22
		PRP	19
	Rule parameter changing operator	RRD	60
		CRD	60
	Hierarchy changing operators	RPD	5
		APD	5
Rule adding operator		ANR	200
Total			371



Results

	Functiona I	CR1	CR2
#Test Cases	42	20	35
Overall score with basic security operators	78%	87%	100%
Rule adding operator (ANR)	11%	14%	17%
Overall score with all operators	42%	47%	55%



➔ Functional tests vs. basic security ones

➤ Functional test cases:

- 78% of basic security mutants
- 11% of ANR
- 7 of the 42 test cases do not trigger security mechanisms
- Functional tests \leftrightarrow Security Policy tests

➤ CR2 > CR1 :

- 100% MS with basic mutation operators

➤ None of these criteria are sufficient to kill ANR mutants



➔ Advanced vs. basic security tests

	#test cases	Basic security mutants	ANR
CR2	35	100%	17%
Advanced sec. tests	154	59.3%	100%

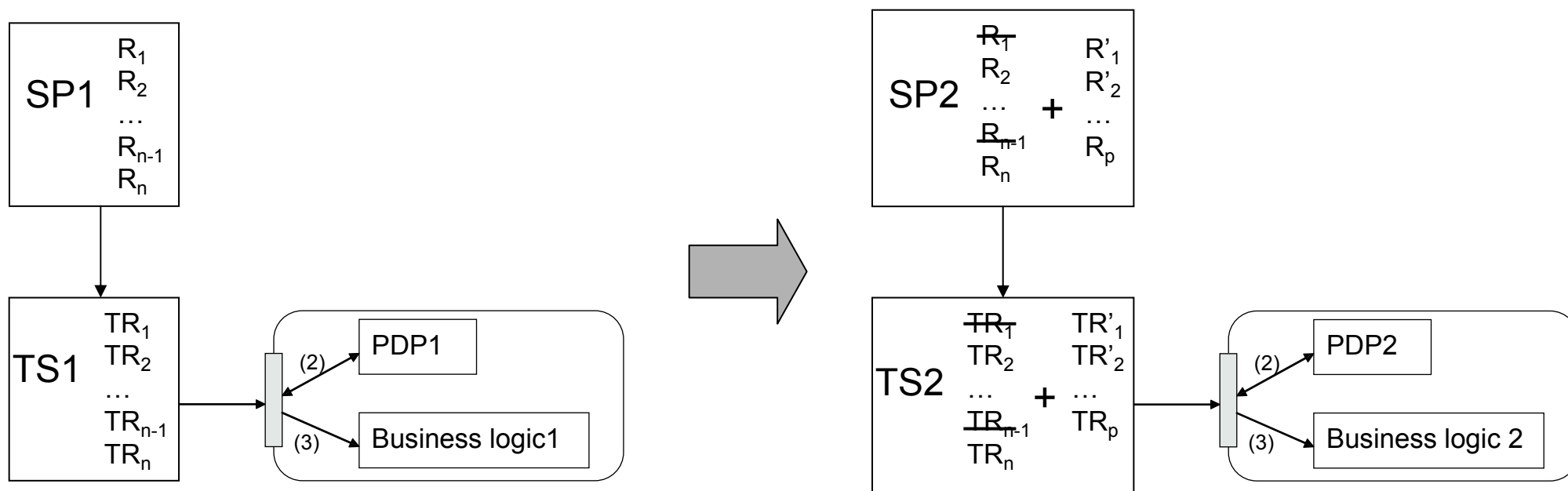
- Advanced security test cases kill all ANR mutants
 - **a costly task**
- But fail in killing all basic security mutants
- Both criteria are thus needed to be fully efficient in testing the security mechanisms

➔ Reusing functional test cases ?

Strategy	Funct .	Basic Security CR2	Advanced Security	Total
Independent	-	35	154	189
Incremental from functional	(42)	21	133	196 (154)
Incr. from CR2	-	35	133	168

- Incr. from functional : No clear conclusion
- Incr. from CR2 : 15% of effort saved compared to independent test generation

➔ Regression Testing of the system



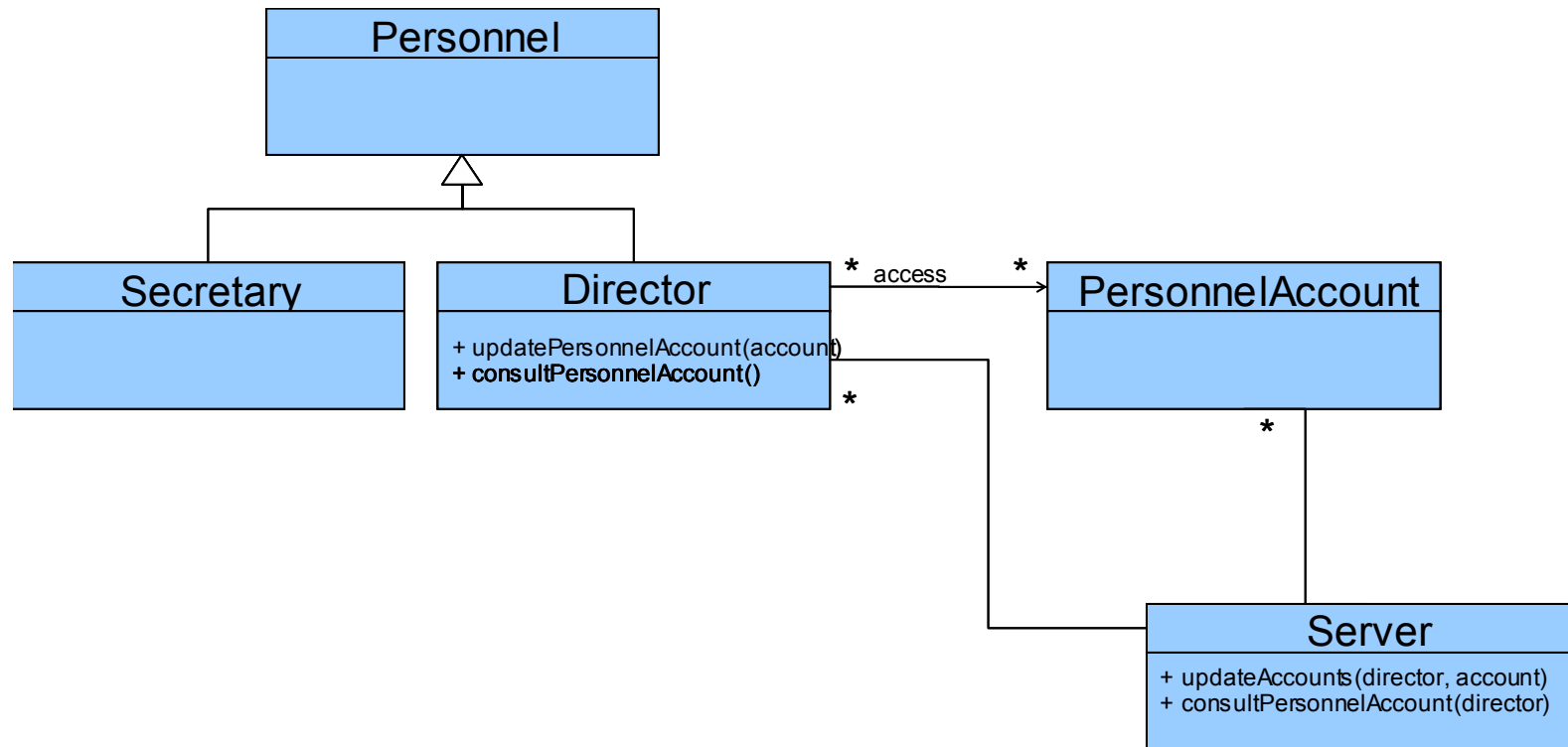
SPi: Security Policy i
 TSi : Test suite i
 PDPi: Policy Decision Point i
 TRi: Test rule i

➔ Explicit/hidden security mechanisms

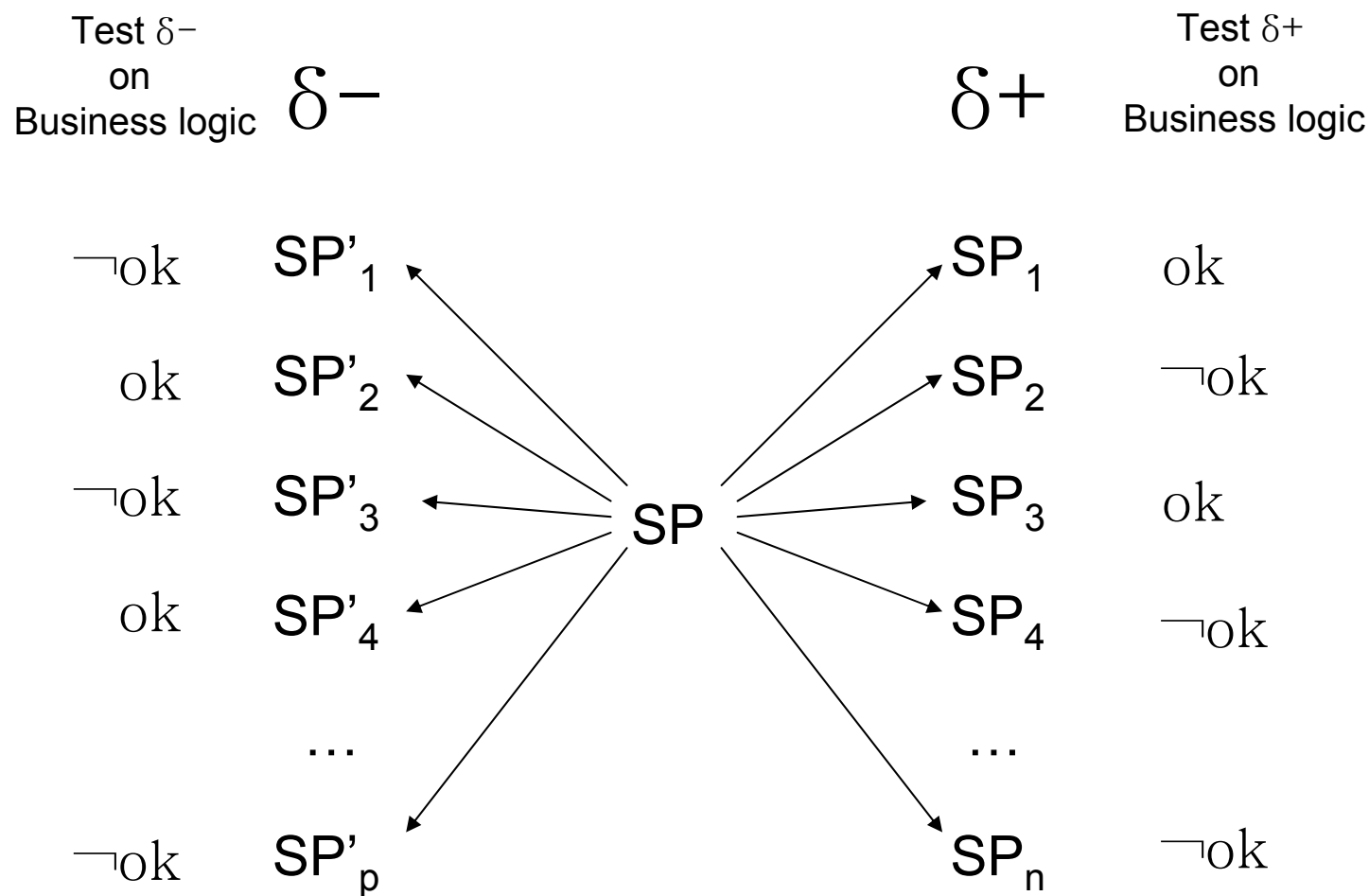
```
public void borrowBook(User user, Book book) throws SecurityPolicyViolationException {
    // call to the security service
    ServiceUtils.checkSecurity(user,
        LibrarySecurityModel.BORROWBOOK_METHOD,
        LibrarySecurityModel.BOOK_VIEW,
        ContextManager.getTemporalContext());
    // call to business objects
    // borrow the book for the user
    book.execute(Book.BORROW, user);
    // call the dao class to update the DB
    bookDAO.insertBorrow(userDTO, bookDTO);}
}
```

```
Public void borrowBook(Book b, User user) {
    // visible mechanism, call to the security policy service
    SecurityPolicyService.check(user,
        SecurityModel.BORROW_METHOD, Book.class, SecurityModel.DEFAULT_CONTEXT);
    // do something else
    // hidden mechanism
    If(getDayOfWeek().equals("Sunday") || getDayOfWeek().equals("Saturday")) {
        // this is not authorized throw a business exception
        Throw new BusinessException("Not allowed to borrow in week-ends);} ...}
}
```

➔ Explicit/hidden security mechanisms



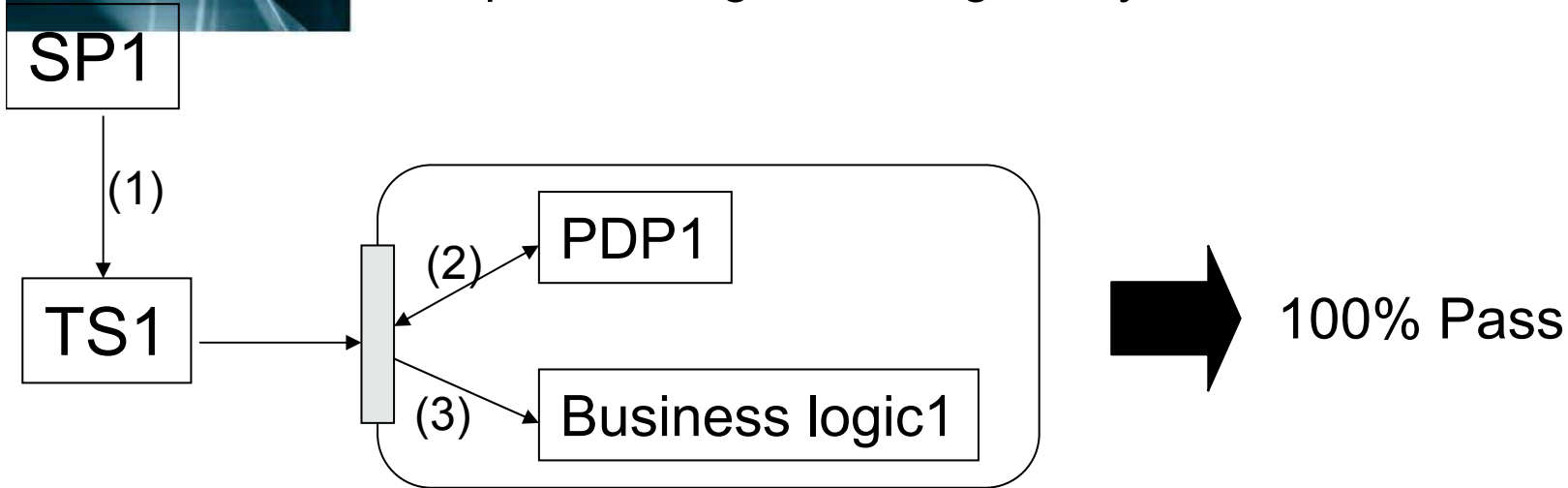
➔ Micro-evolutions of a legacy system access control policy



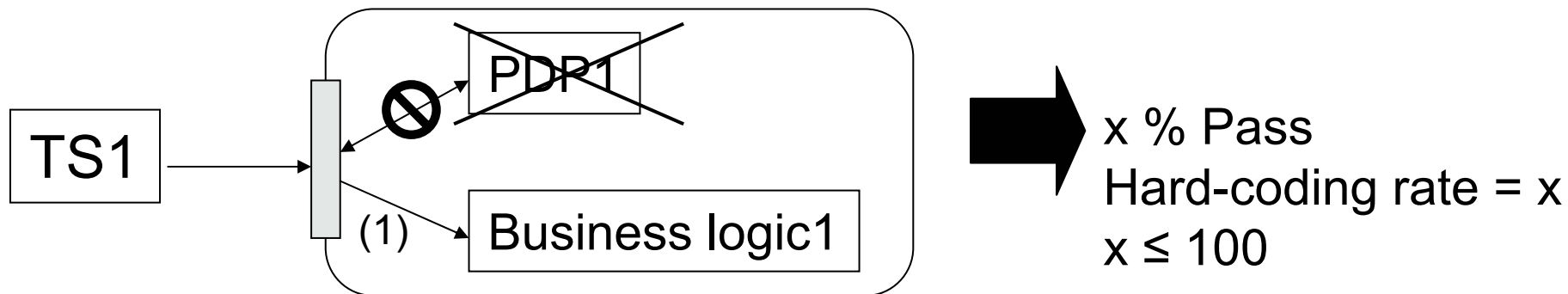
δ^- = access control rule removal
 δ^+ = access control rule addition

➔ Test driven audit of the current system

Step 1 : testing/correcting the system w.r.t. the security policy



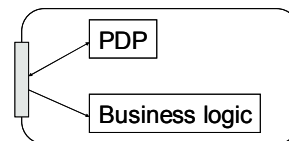
Step 2 : reapply test cases to check the security policy « hard-coding » rate



SP_i: Security Policy *i*

TS_i : Test suite *i*

PDP_i: Policy Decision Point *i*



: a legacy system

➔ Conclusion

- A qualification process
- A first result : specific testing techniques are needed to guarantee the coverage of access control aspects
- Optimize mutation for security
- Automate the security policy test generation (Advanced security test cases)



➔ **Questions?**