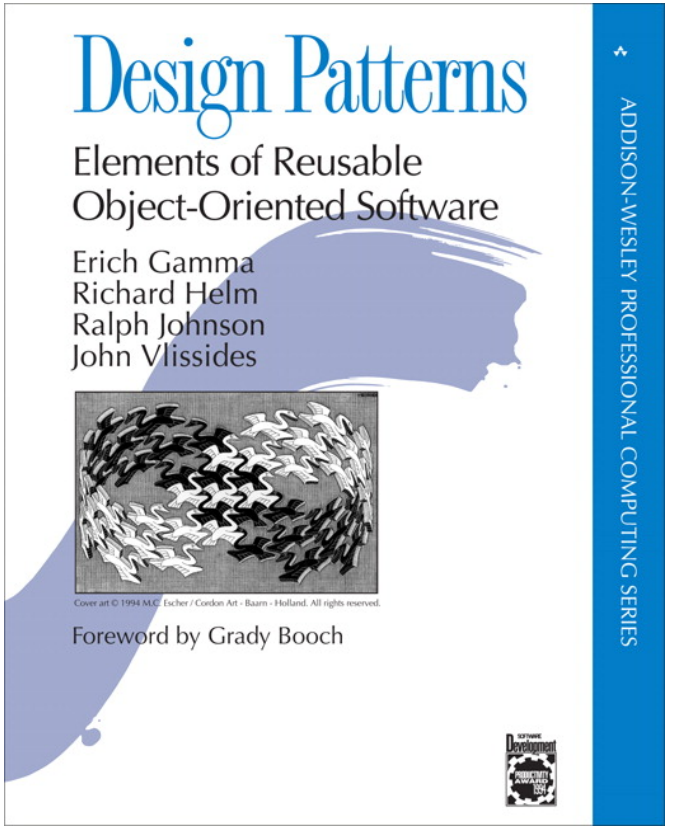
Variability-Aware Design Patterns

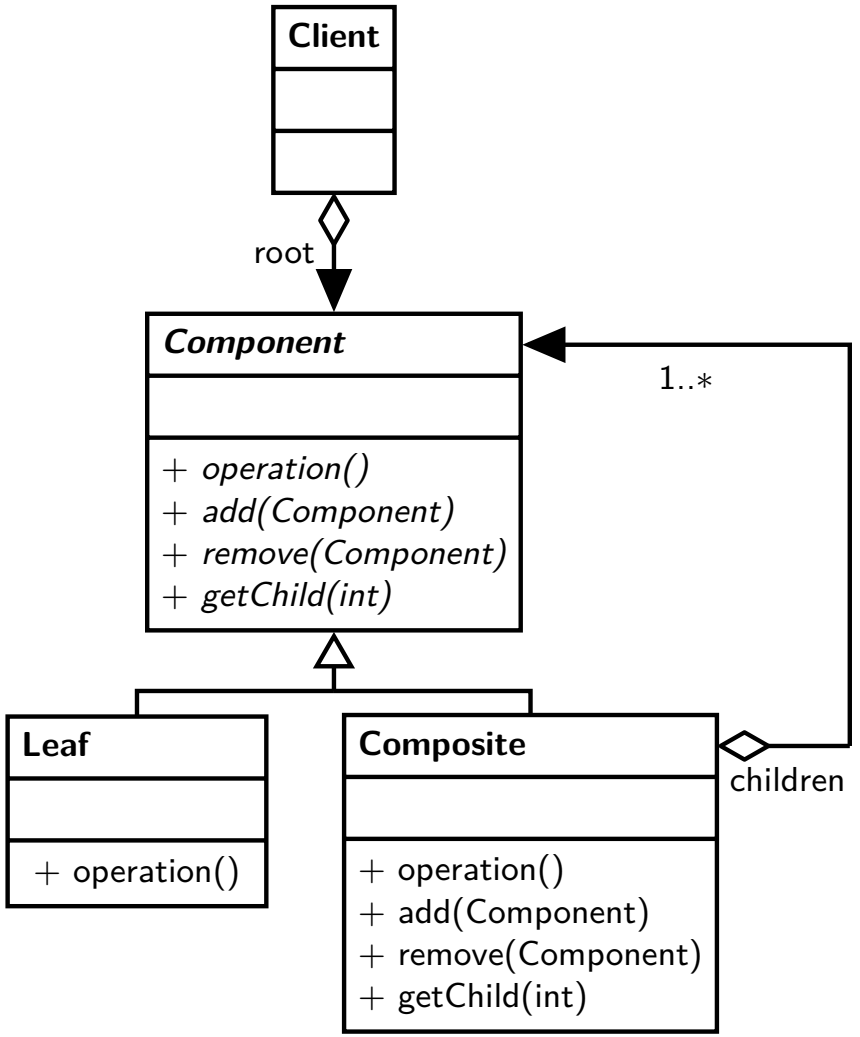
WG 2.11 – Program Generation – Stellenbosch, South Africa

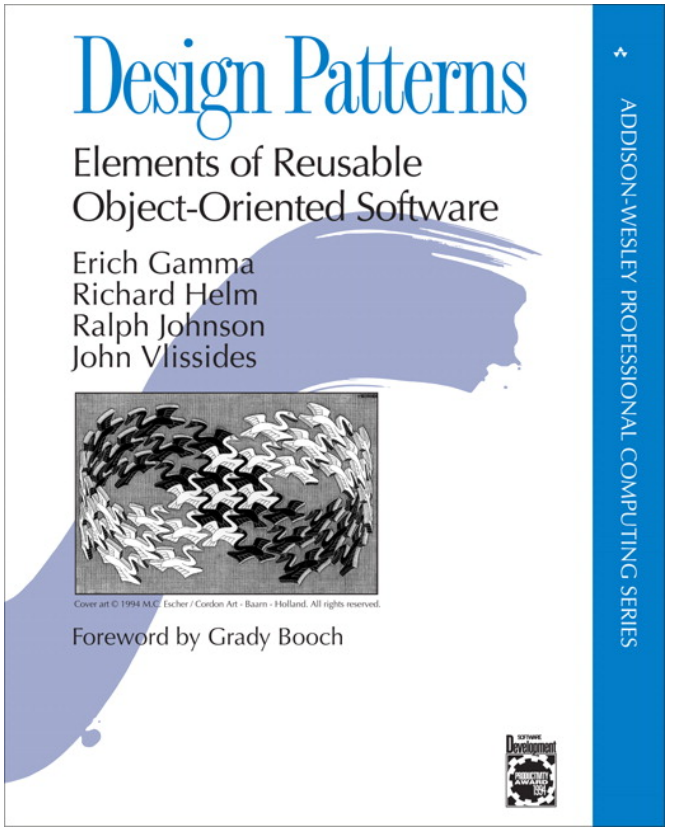
Ina Schaefer, Sven Schuster, Christoph Seidl, January 20 – 22, 2015

Design Patterns

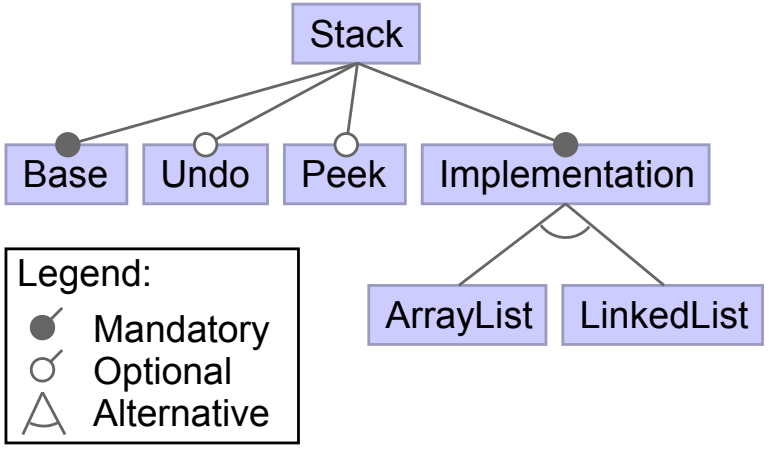


Design Patterns





Feature-oriented Programming



Feature Base

```
class Stack
  void push(int v)
    { /* ... */ }
  int pop() { /* ... */ }
}
```

Feature Undo

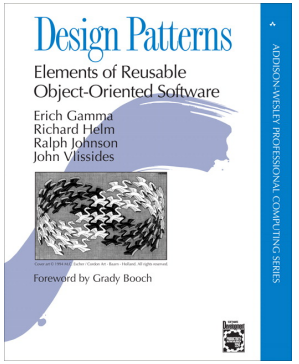
```
class Stack { /* ... */
  int backupPush;
  void undo() { /* ... */ }
  void push(int v) {
    backupPush=v;
    original(v);
  }
}
```

Superimposition

```
class Stack { /* ... */
  int backupPush;
  int pop() { /*...*/ }
  void undo() { /*...*/ }
  void push(int v) {
    backupPush=v;
    /* original push */
  }
}
```

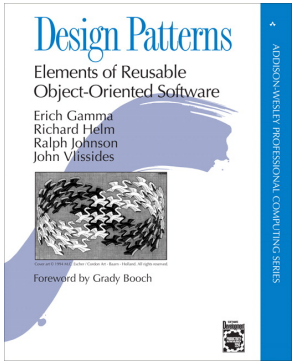
Motivation

- Design Patterns are general design solutions



Motivation

- Design Patterns are general design solutions
- FOP introduces new layer of design



Feature Base

```
class Stack
  void push(int v)
    { /* ... */ }
  int pop() { /* ... */ }
}
```

Feature Undo

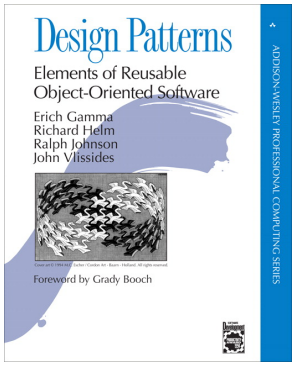
```
class Stack { /* ... */
  int backupPush;
  void undo() { /* ... */ }
  void push(int v) {
    backupPush=v;
    original(v);
  }
}
```

Superimposition

```
class Stack { /* ... */
  int backupPush;
  int pop() { /*...*/ }
  void undo() { /*...*/ }
  void push(int v) {
    backupPush=v;
    /* original push */
  }
}
```

Motivation

- Design Patterns are general design solutions
- FOP introduces new layer of design
- Does it blend?



Feature Base

```
class Stack
  void push(int v)
    { /* ... */ }
  int pop() { /* ... */ }
}
```

Feature Undo

```
class Stack { /* ... */
  int backupPush;
  void undo() { /* ... */ }
  void push(int v) {
    backupPush=v;
    original(v);
  }
}
```

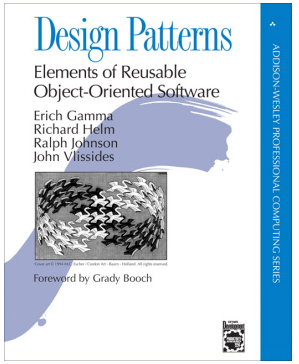
Superimposition

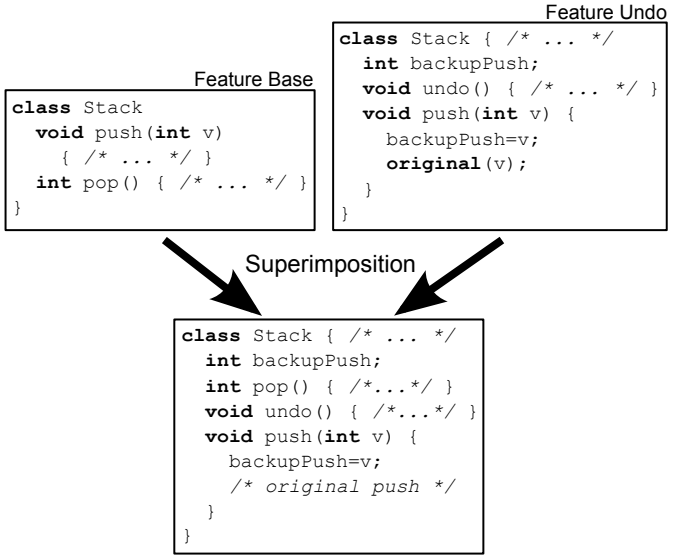
```
class Stack { /* ... */
  int backupPush;
  int pop() { /*...*/ }
  void undo() { /*...*/ }
  void push(int v) {
    backupPush=v;
    /* original push */
  }
}
```

Motivation

- Design Patterns are general design solutions
- FOP introduces new layer of design

- Does it blend?
  - Yes, it does!
  - Decomposed patterns
    * Schuster,Schulze,Schaefer@VaMoS'14



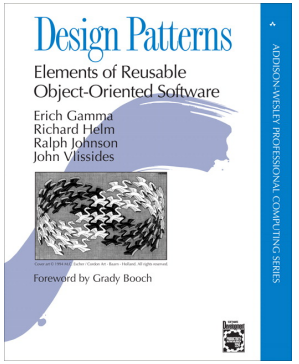


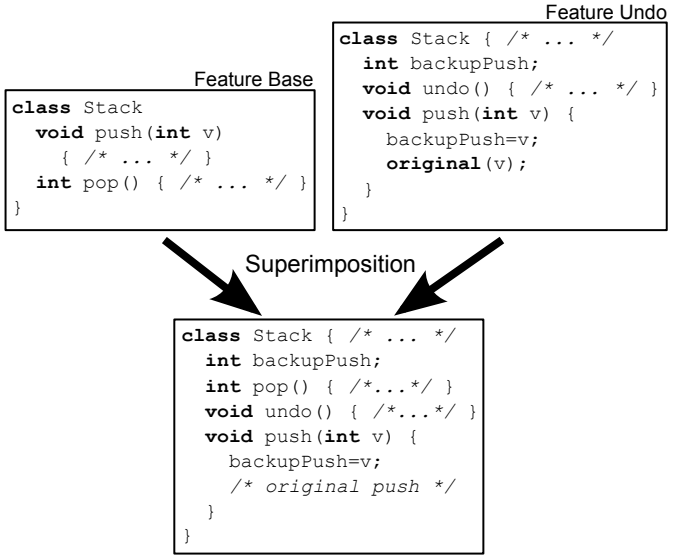
Motivation

- Design Patterns are general design solutions
- FOP introduces new layer of design

- Does it blend?
  - Yes, it does!
  - Decomposed patterns
    * Schuster,Schulze,Schaefer@VaMoS'14

- But how?





Tasks & Requirements

Comprehensive case study necessary

- Automated family-based detection of patterns in FOP
- Family-based analysis of detected patterns
- Role Modeling to describe family attributes of patterns

Tasks & Requirements

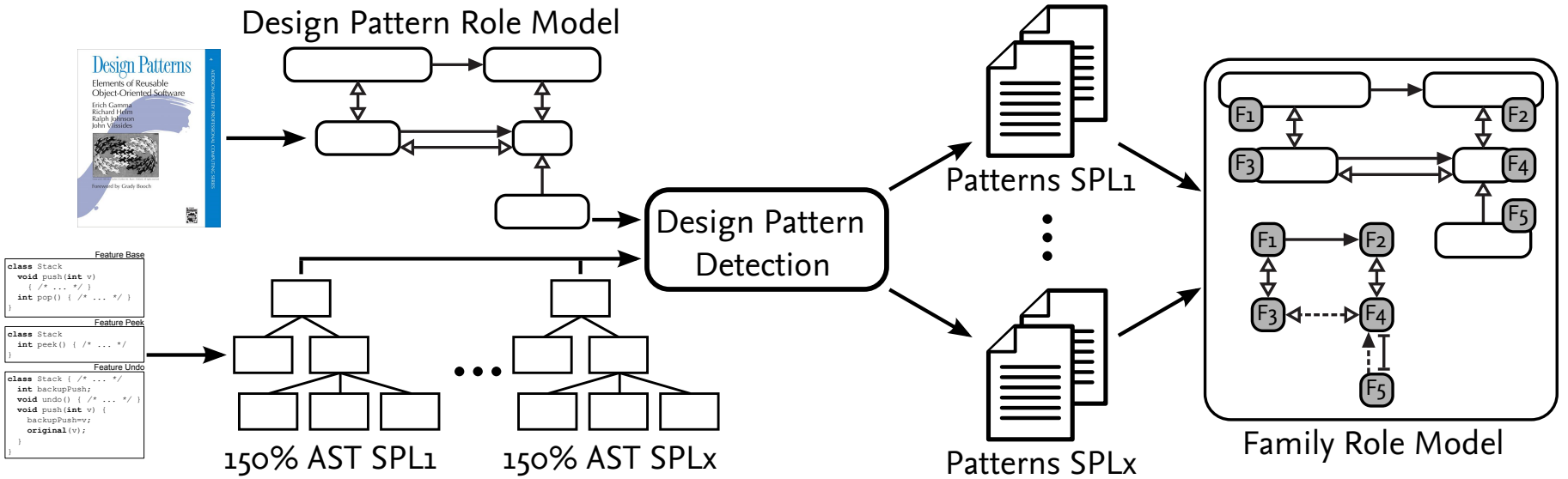
Comprehensive case study necessary

- Automated family-based detection of patterns in FOP
- Family-based analysis of detected patterns
- Role Modeling to describe family attributes of patterns

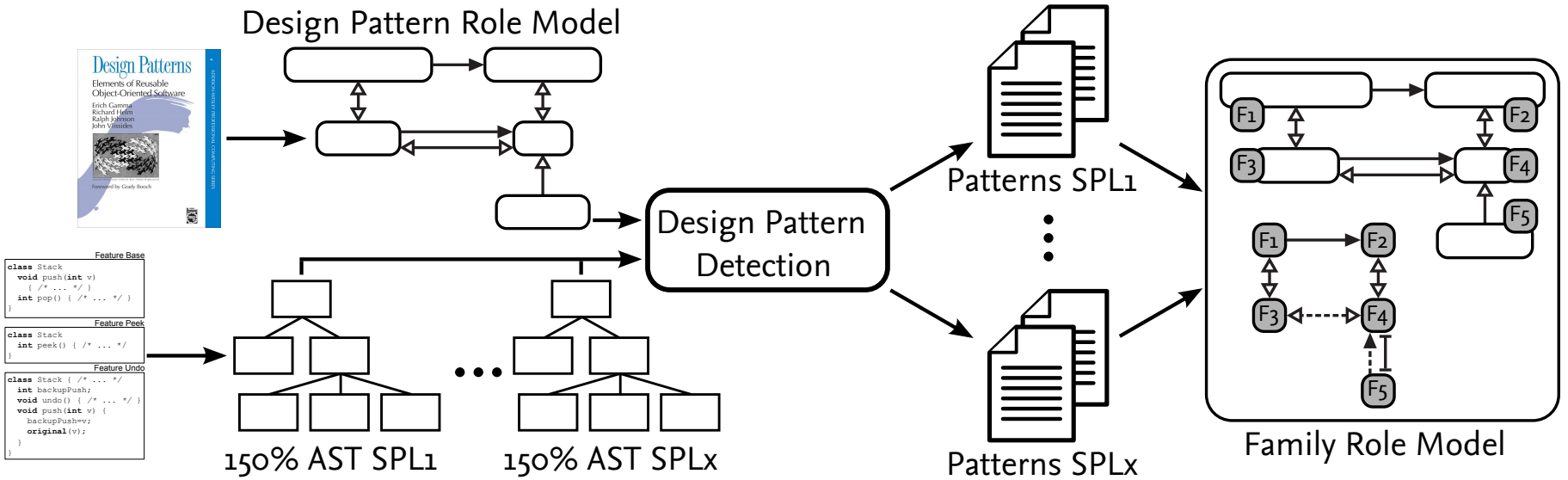
To this end...

1. Parse feature-oriented code
2. Automated design pattern detection
3. Assemble variability-aware design pattern catalog

General Approach



General Approach



1. Role Modeling for Design Patterns
2. Case Study on Design Patterns in SPLs
3. Variability-Aware Design Pattern Catalog
   - Introducing Family Role Models

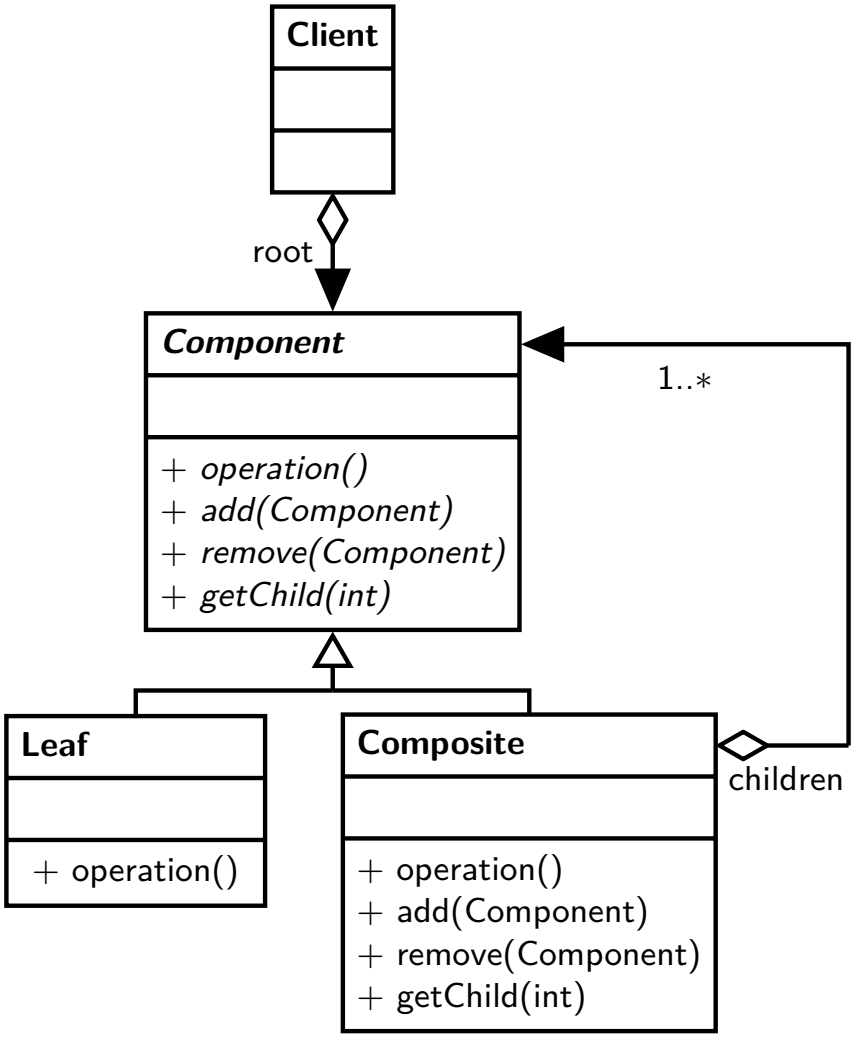
Part I

Role Modeling – Describing Dynamic Collaborations of Design Patterns

Role Modeling – Motivation

Describing Design Patterns

- Design Patterns commonly described with class diagrams
- Class diagrams = static structure
- But design patterns = dynamic behavior / collaborations

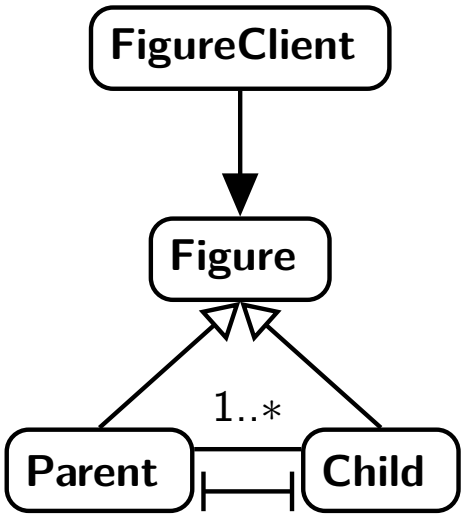


Role Modeling

Role Modeling (Reenskaug, 1996)

- Describe collaborations of roles
- Dynamic behavior vs. static structure
- Objects can play various roles
- Define constraints between roles
  - uses, implies, equals, prohibition, ...

⇒ Convenient for pattern description!
  - (Dirk Riehle, 1996)



Role Modeling

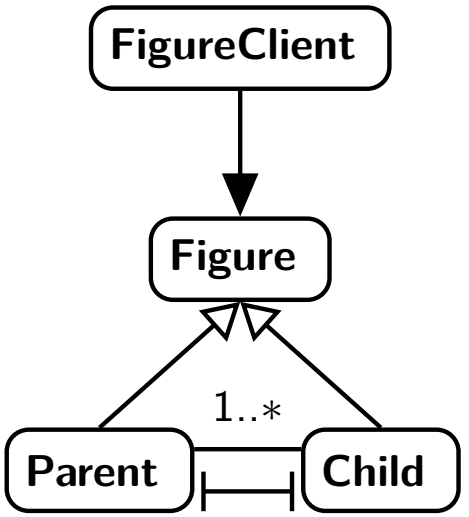
Role Modeling (Reenskaug, 1996)

- Describe collaborations of roles
- Dynamic behavior vs. static structure
- Objects can play various roles
- Define constraints between roles
  - uses, implies, equals, prohibition, ...

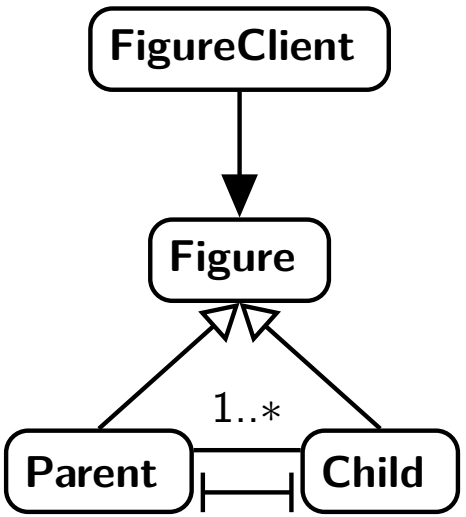
⇒ Convenient for pattern description!

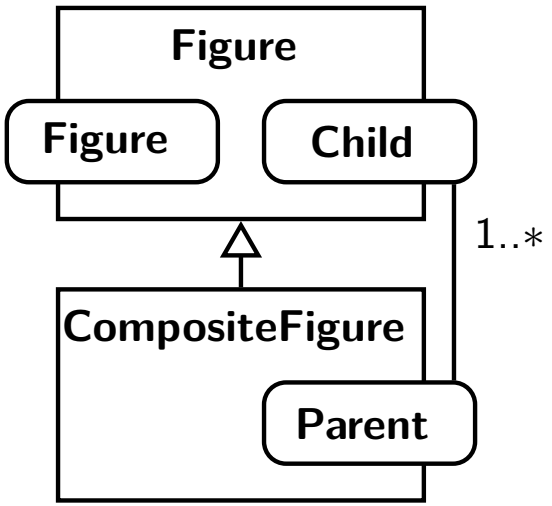
  - (Dirk Riehle, 1996)

- Independent of mapping
  - Object-oriented design
  - Feature models
  - ...



Mapping of Roles to Classes





Part II

Case Study – Detecting Design Patterns in Feature-Oriented SPLs

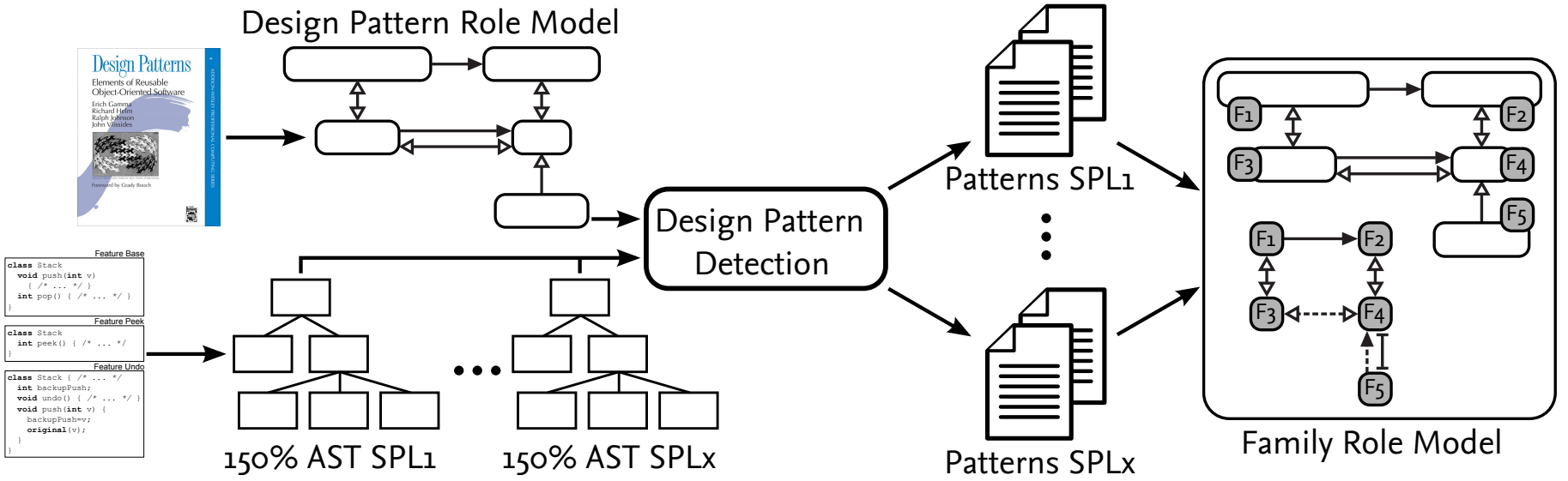
Case Study – Research Questions

1. *How are design pattern implementations decomposed along features?*

2. *How are the involved features related to each other?*

3. *In what variability-aware fashion are the analyzed design patterns commonly applied?*

Case Study – Methodology



Case Study – Setup

Setup Requirements

- Representative set of modularity & variability patterns
- SPLs of different sizes & domains

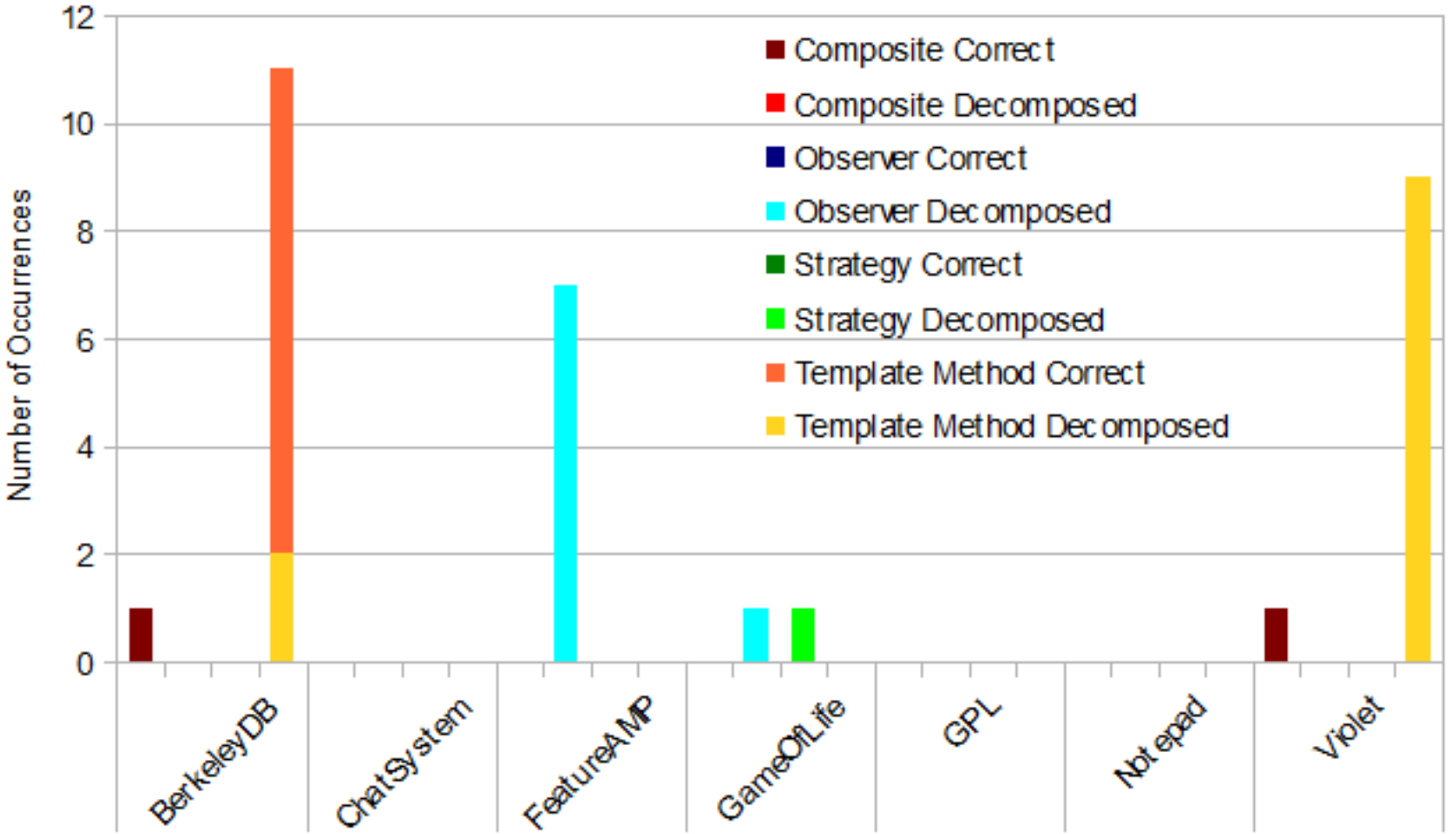
Design Patterns

- Composite
- Observer
- Objectifier / Strategy
- Template Method

Product Lines

- BerkeleyDB (45K)
- ChatSystem (1K)
- FeatureAMP (2K)
- GameOfLife (1.5K)
- GPL (2K)
- Notepad (1.75K)
- Violet (7.5K)

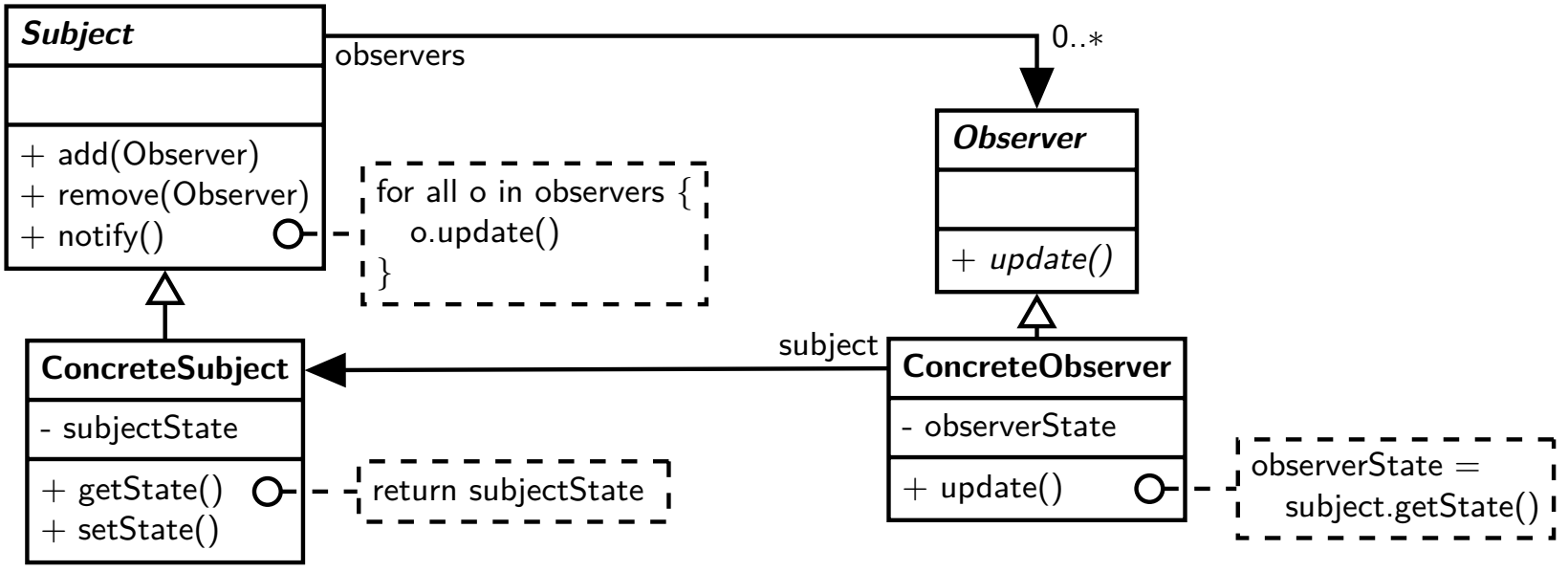
Case Study – Results – Overview



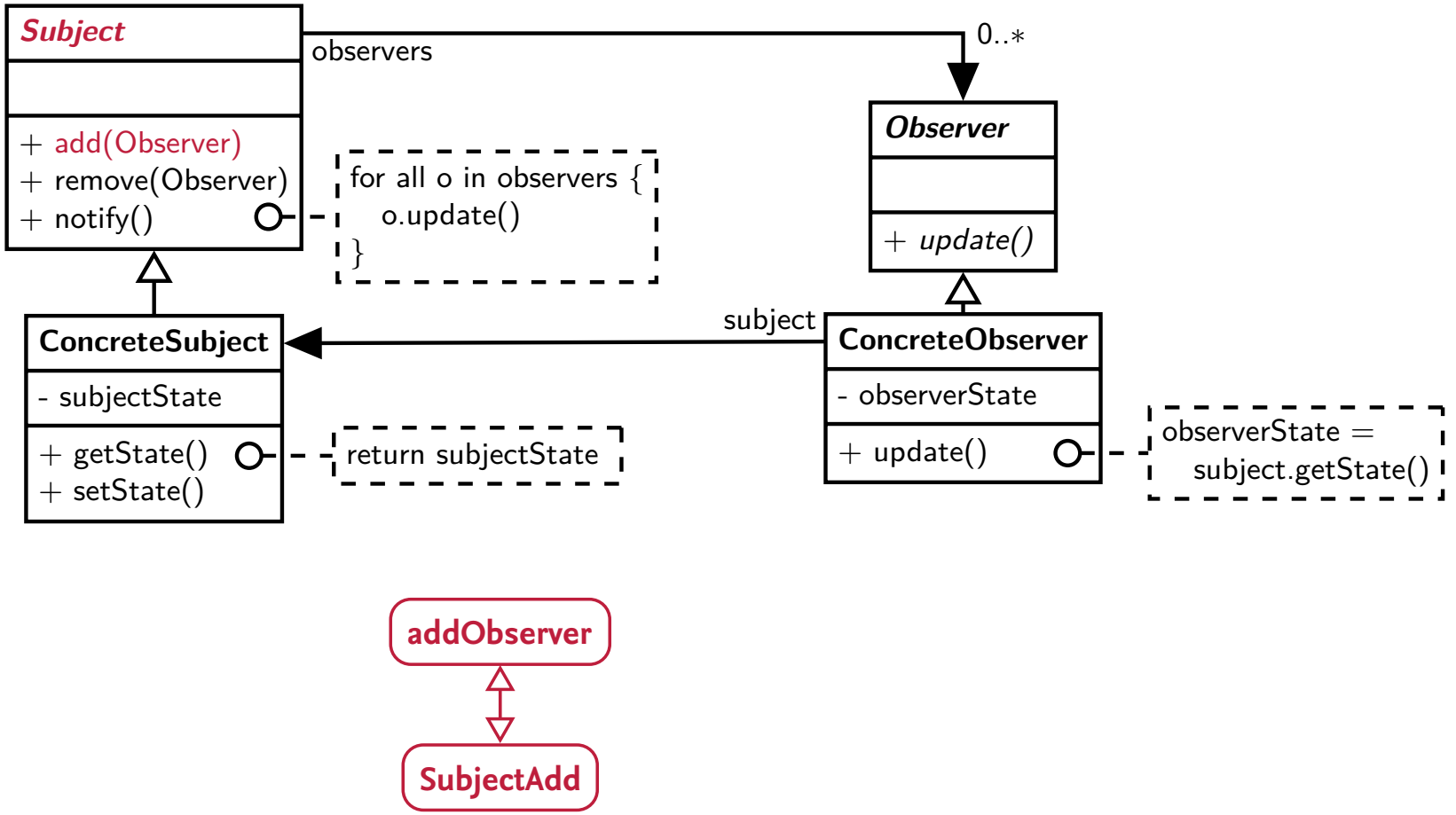
Answering Research Questions

1. *How are design pattern implementations decomposed along features?*

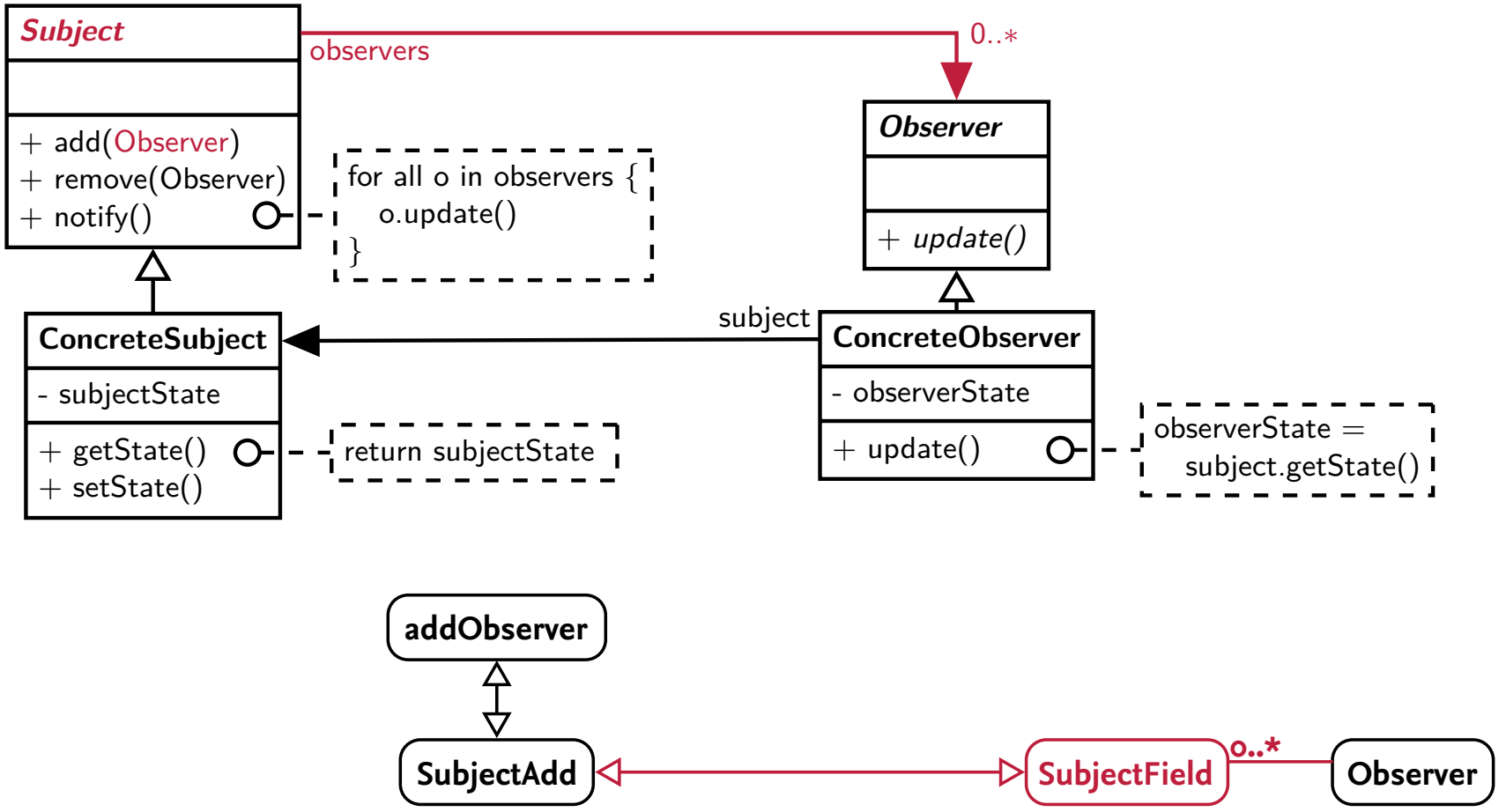
Observer pattern



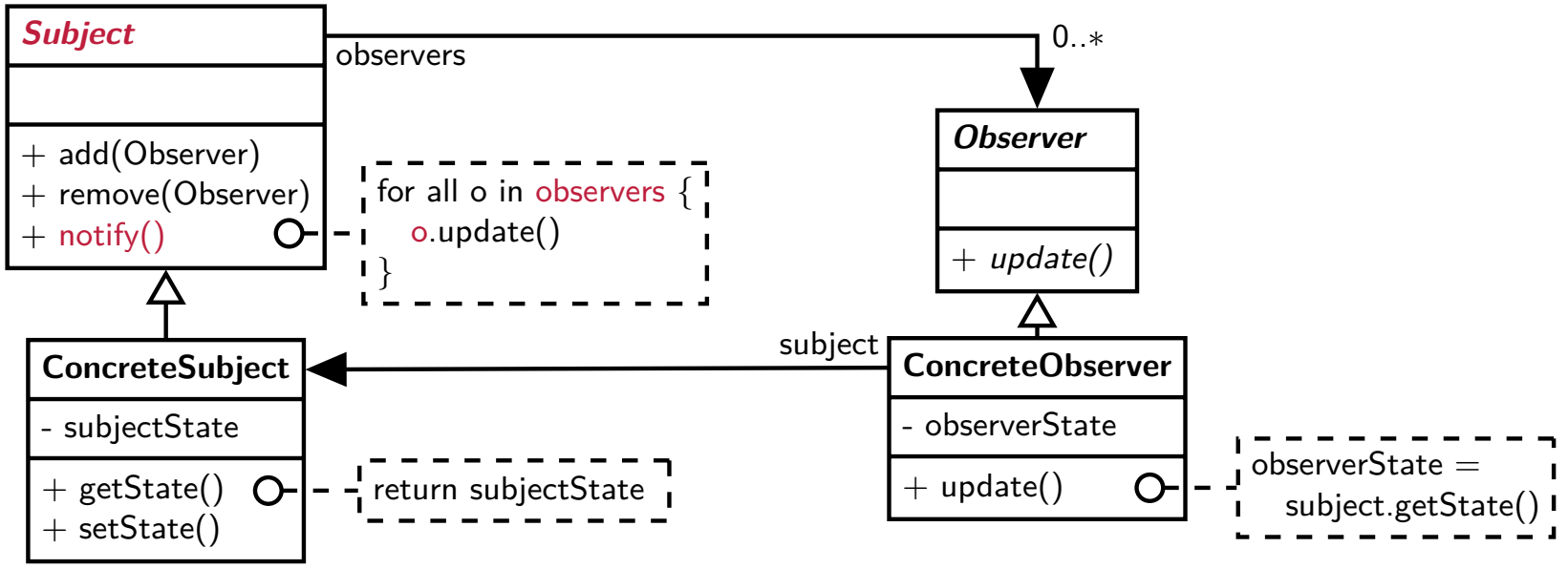
Observer pattern

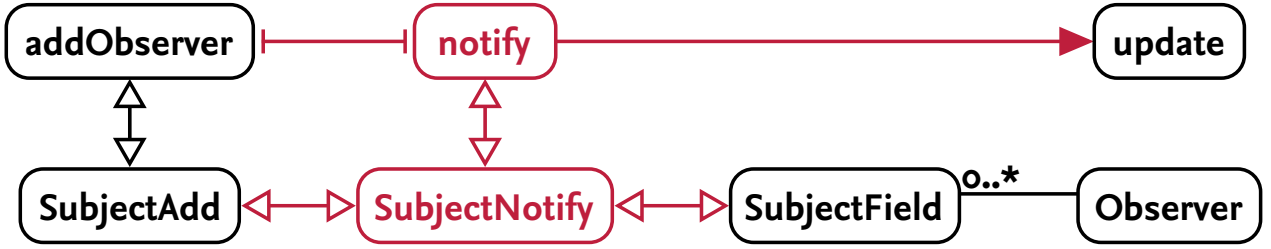


Observer pattern

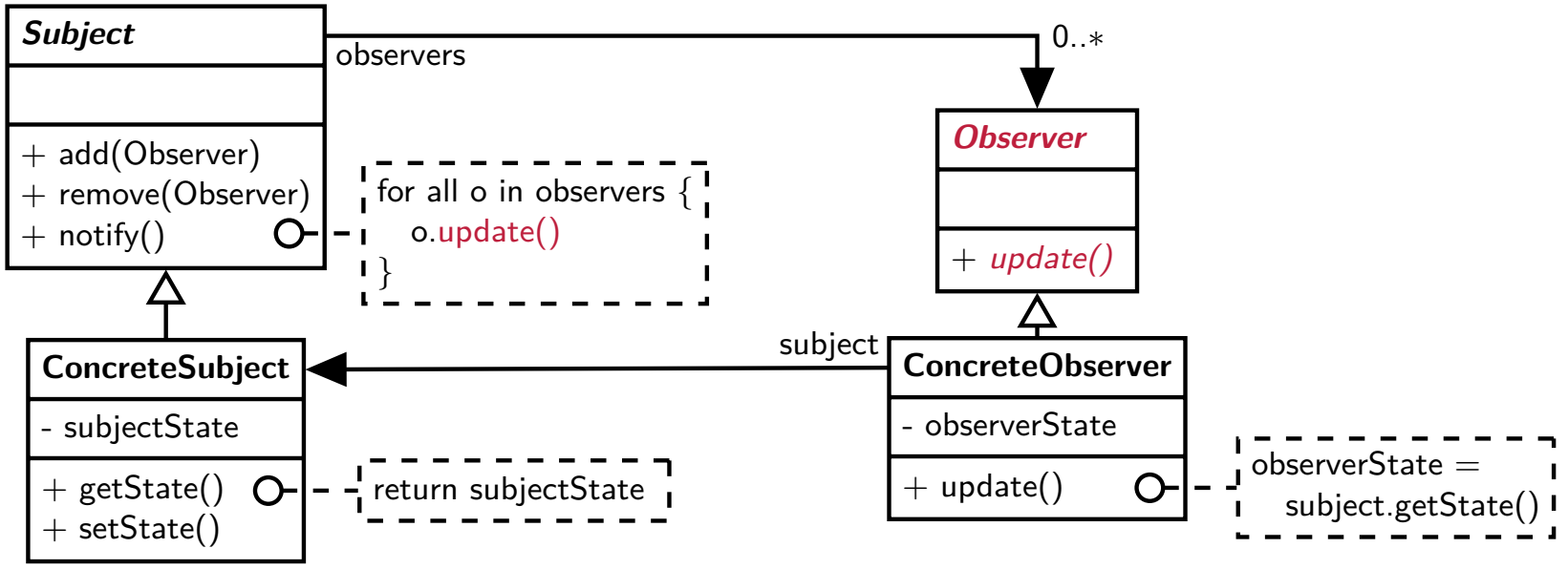


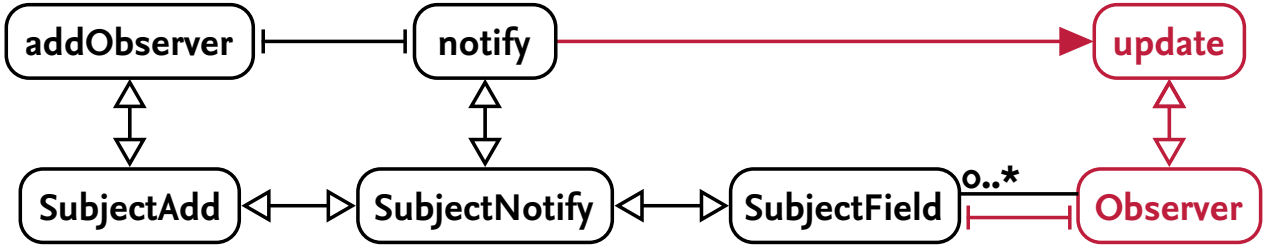
Observer pattern



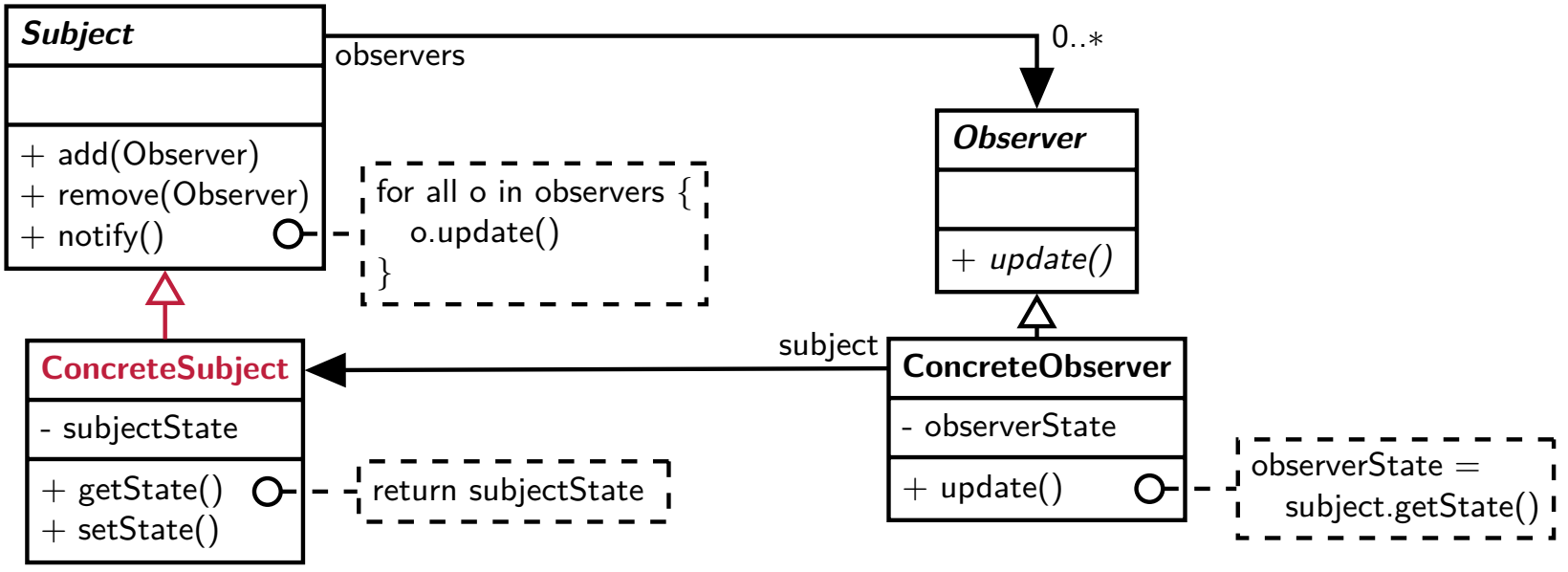


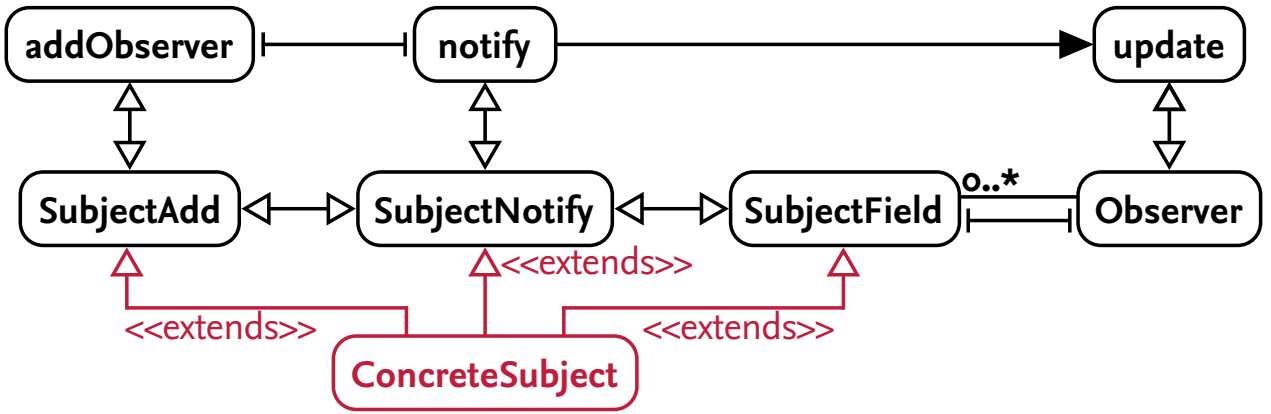
Observer pattern



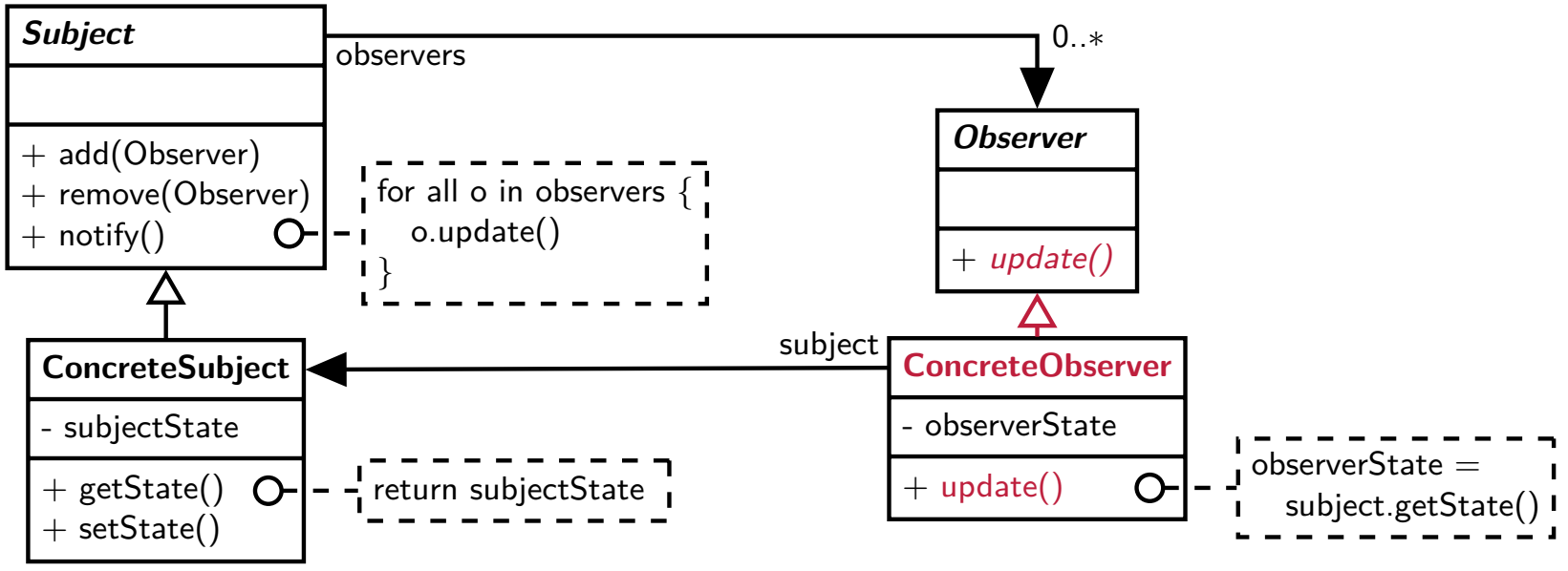


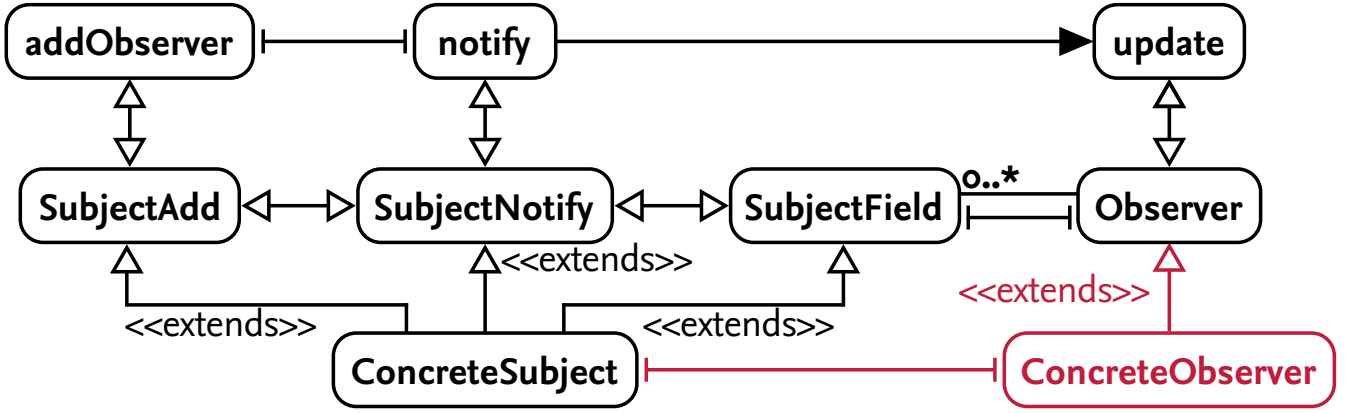
Observer pattern





Observer pattern





Observer in FeatureAMP (1/2)

Feature *BASE_FEATUREAMP*

```
1  public interface Listener<T> { public void update(T object); }
2  public abstract class AbstractAudioController implements AudioController {
3    protected LinkedList<Listener<AudioController>> playListeners;
4    public void addPlayListener(Listener<AudioController> l) {
5      this.playListeners.add(l);
6    }
7    protected void notifyPlayListeners() {
8      for (Listener<AudioController> l: this.playListeners) {
9        l.update(this);
10     }
11   }
12 }
```

Observer in FeatureAMP (1/2)

Feature *BASE_FEATUREAMP*

```
1  public interface Listener<T> { public void update(T object); }
2  public abstract class AbstractAudioController implements AudioController {
3    protected LinkedList<Listener<AudioController>> playListeners;
4    public void addPlayListener(Listener<AudioController> l) {
5      this.playListeners.add(l);
6    }
7    protected void notifyPlayListeners() {
8      for (Listener<AudioController> l: this.playListeners) {
9        l.update(this);
10     }
11   }
12 }
```

Observer in FeatureAMP (1/2)

Feature *BASE_FEATUREAMP*

```
1 public interface Listener<T> { public void update(T object); }
2 public abstract class AbstractAudioController implements AudioController {
3   protected LinkedList<Listener<AudioController>> playListeners;
4   public void addPlayListener(Listener<AudioController> l) {
5     this.playListeners.add(l);
6   }
7   protected void notifyPlayListeners() {
8     for (Listener<AudioController> l: this.playListeners) {
9       l.update(this);
10    }
11  }
12 }
```

Feature *MP3*

```
13 public class Mp3Controller extends AbstractAudioController {
14   public void play() { /* ... */
15     this.notifyPlayListeners();
16   }
17 }
```

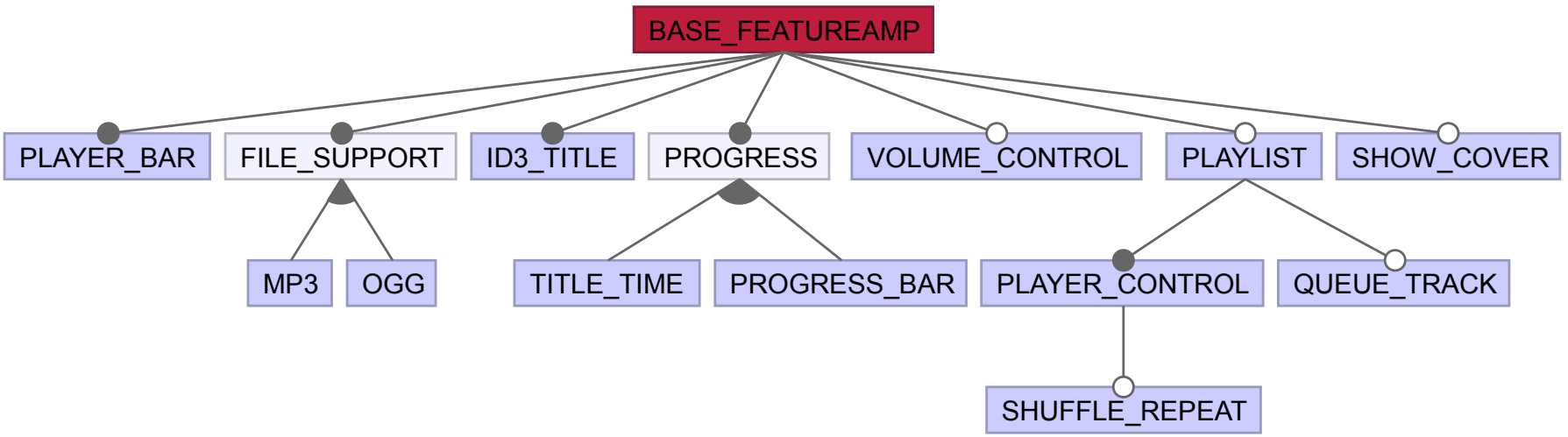
Answering Research Questions

1. *How are design pattern implementations decomposed along features?*

2. *How are the involved features related to each other?*

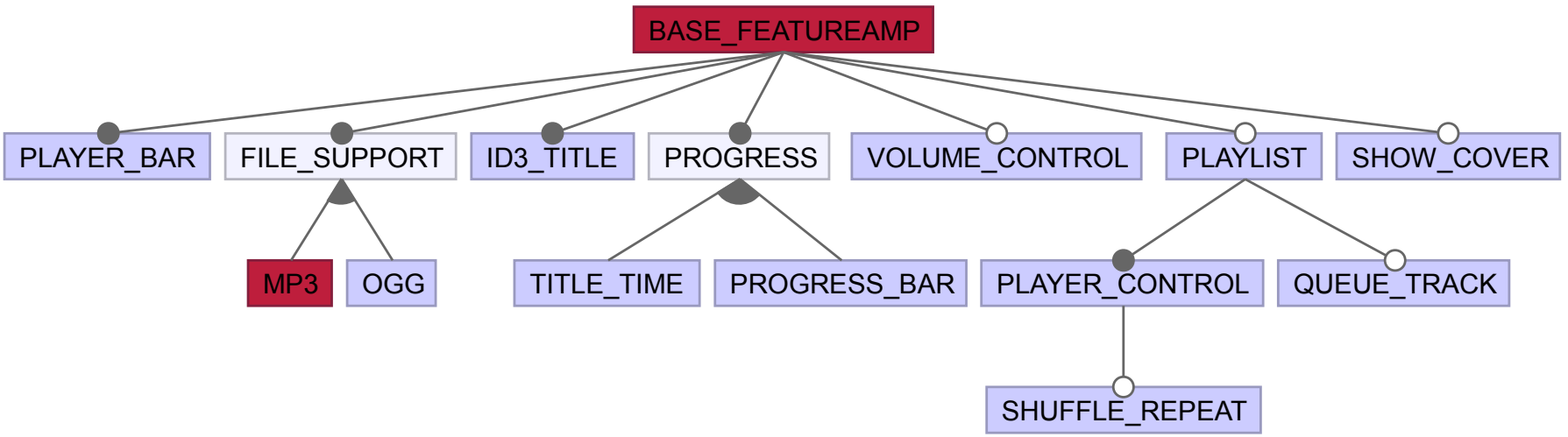
Observer in FeatureAMP (2/2)

Subject & Observer



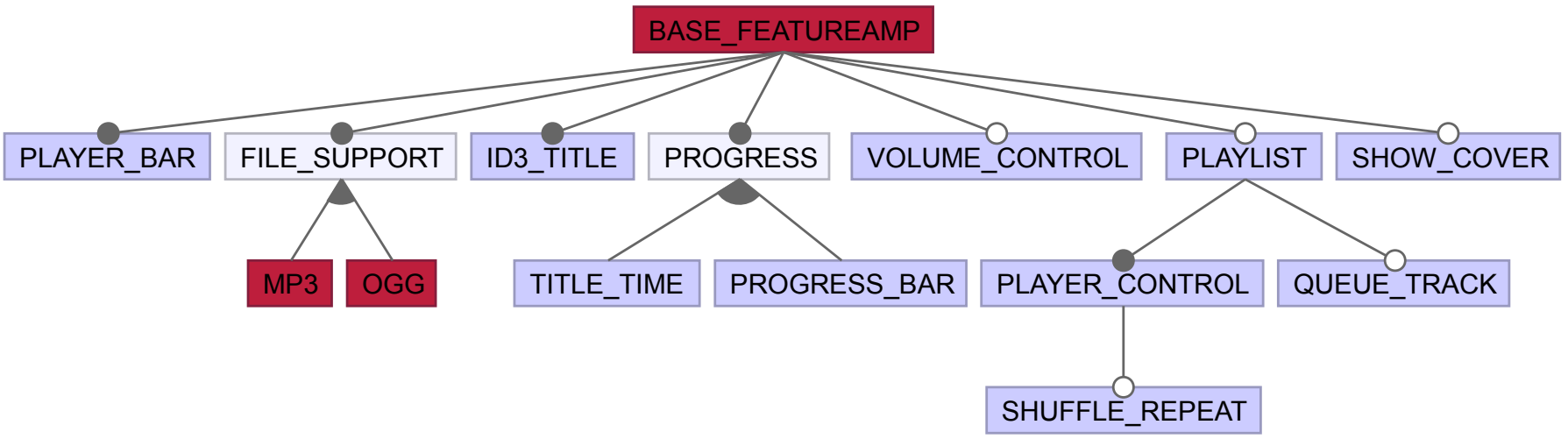
Observer in FeatureAMP (2/2)

ConcreteSubject



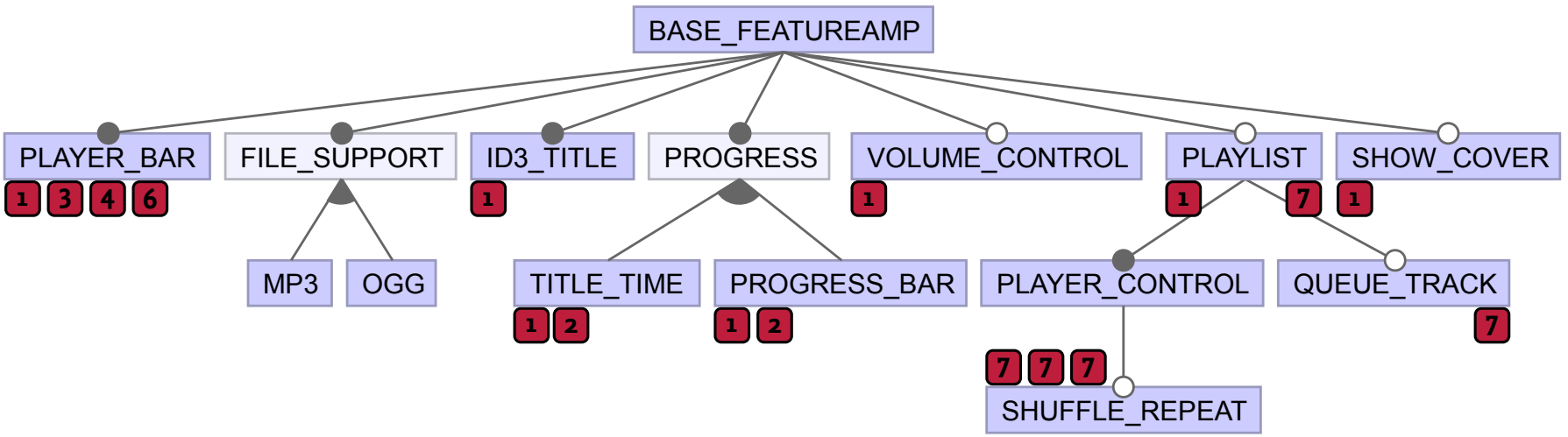
Observer in FeatureAMP (2/2)

ConcreteSubject



Observer in FeatureAMP (2/2)

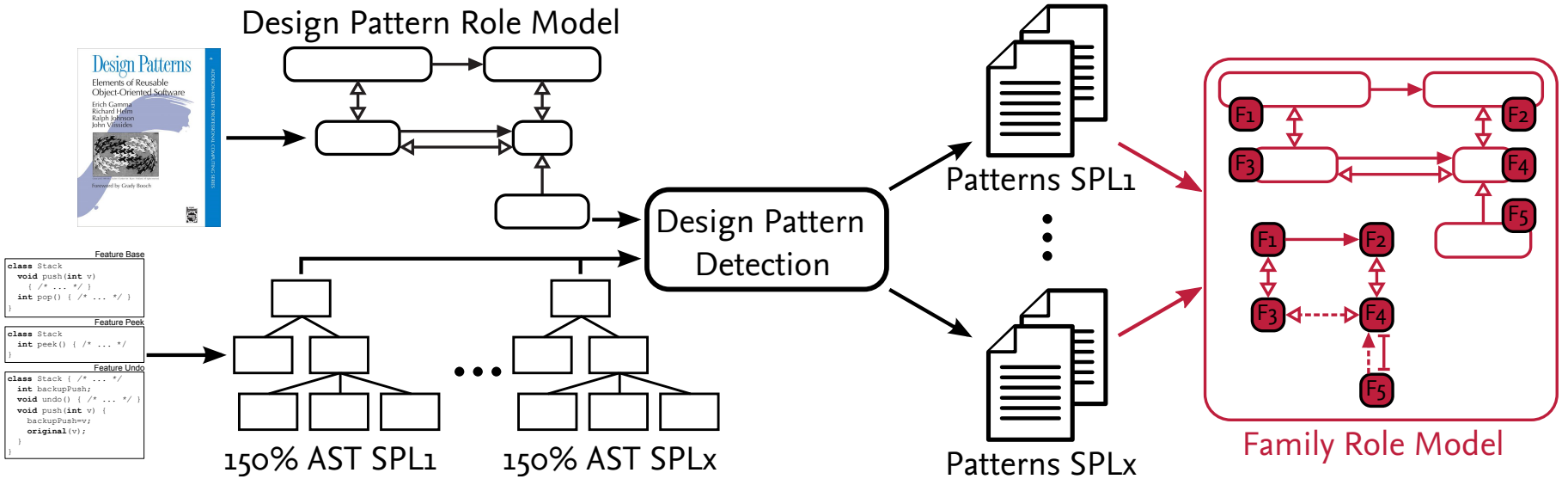
ConcreteObservers



Answering Research Questions

1. *How are design pattern implementations decomposed along features?*
2. *How are the involved features related to each other?*
3. *In what variability-aware fashion are the analyzed design patterns commonly applied?*

Capturing Decomposed Design Patterns



Part III

Family Role Models – Capturing Decomposed Design Patterns

Family Role Models

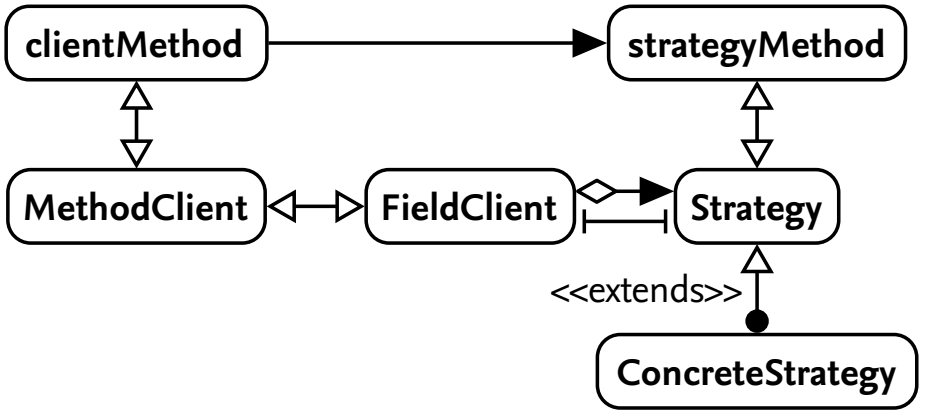
Problem

- Design patterns decomposed along features
- How to describe decomposition?

Family Role Models

Problem

- Design patterns decomposed along features
- How to describe decomposition?



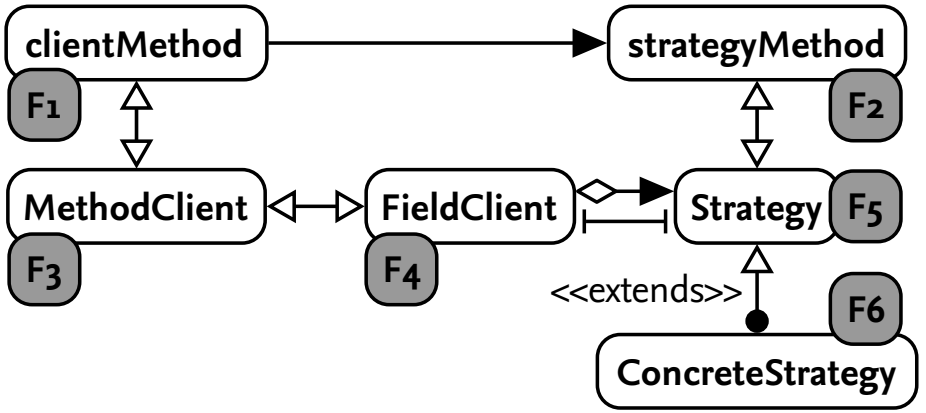
Family Role Models

- Role model to capture pattern

Family Role Models

Problem

- Design patterns decomposed along features
- How to describe decomposition?



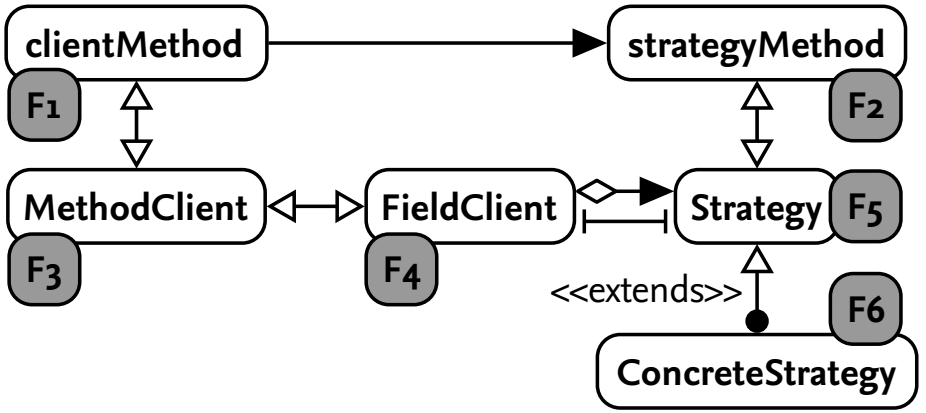
Family Role Models

- Role model to capture pattern
- Annotate with feature roles

Family Role Models

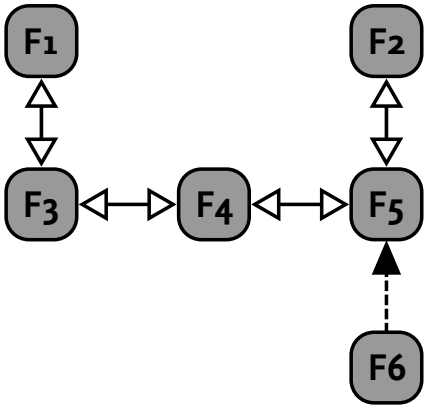
Problem

- Design patterns decomposed along features
- How to describe decomposition?

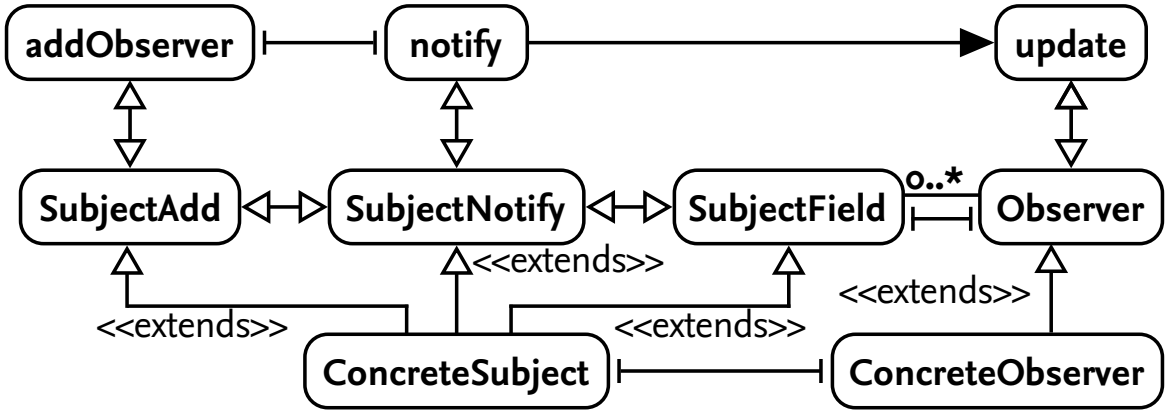


Family Role Models

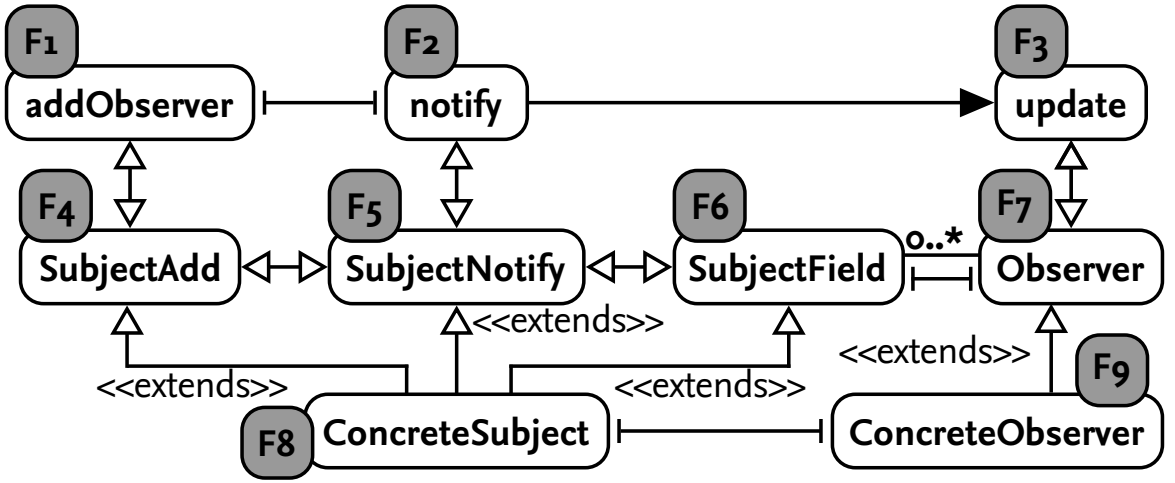
- Role model to capture pattern
- Annotate with feature roles
- Role model to capture feature collaborations



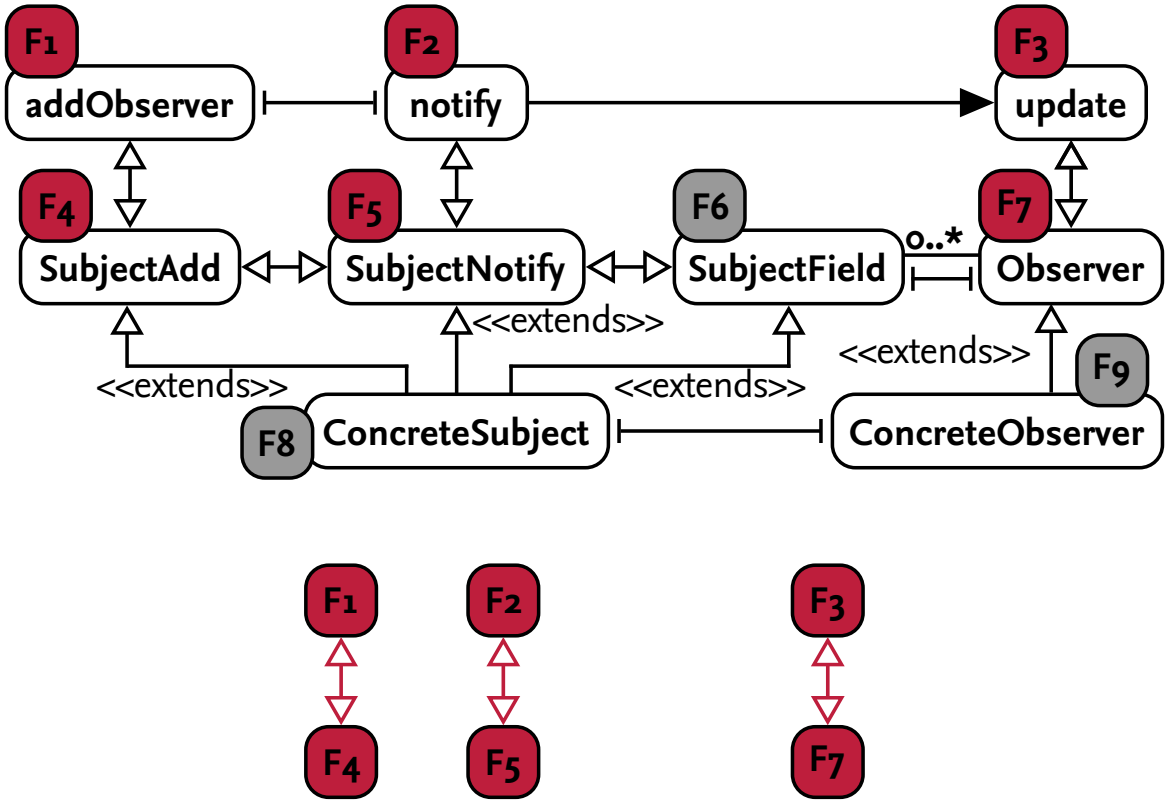
Family Role Model of Observer Pattern



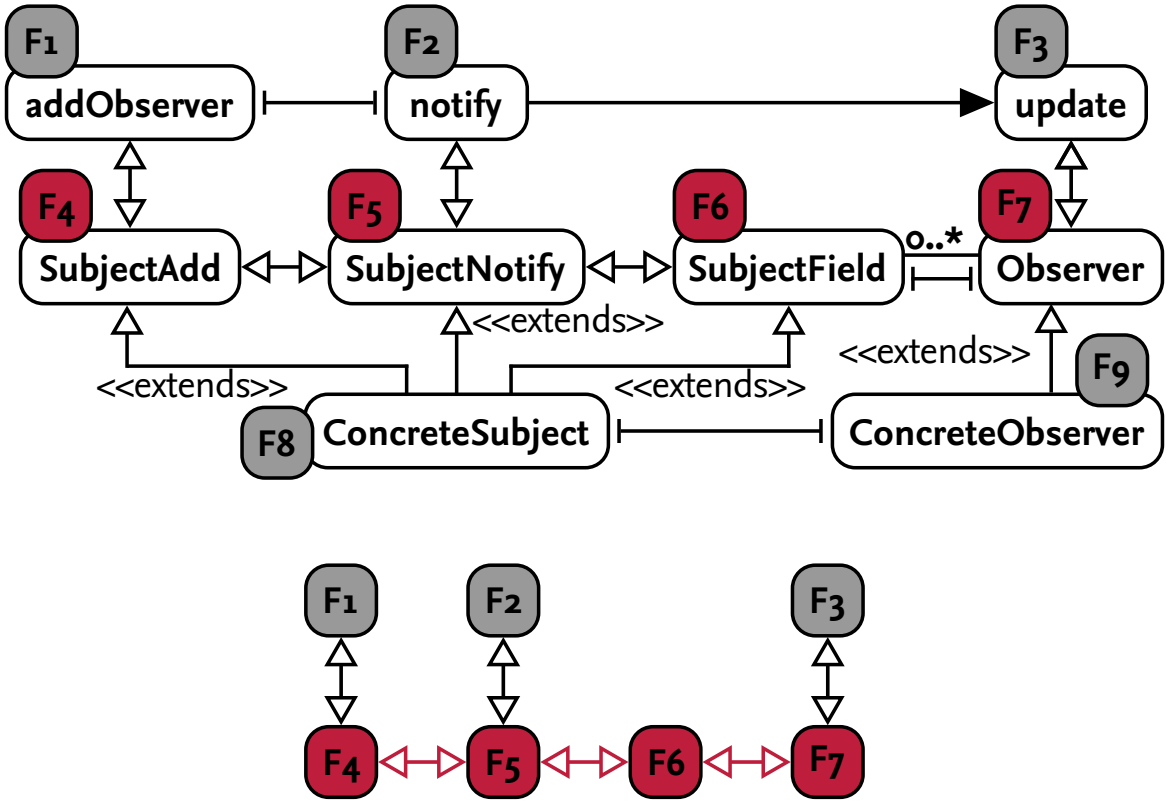
Family Role Model of Observer Pattern



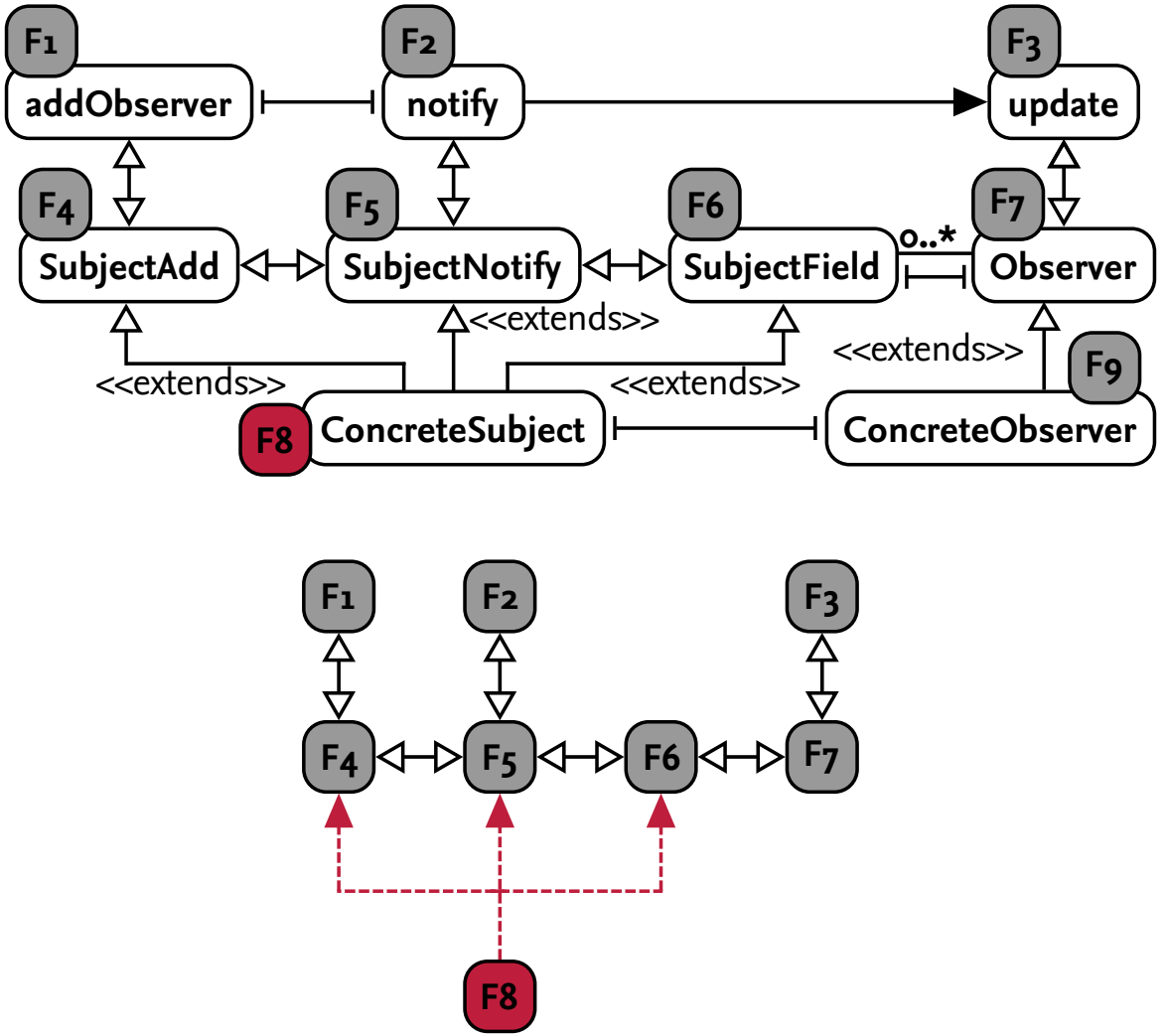
Family Role Model of Observer Pattern



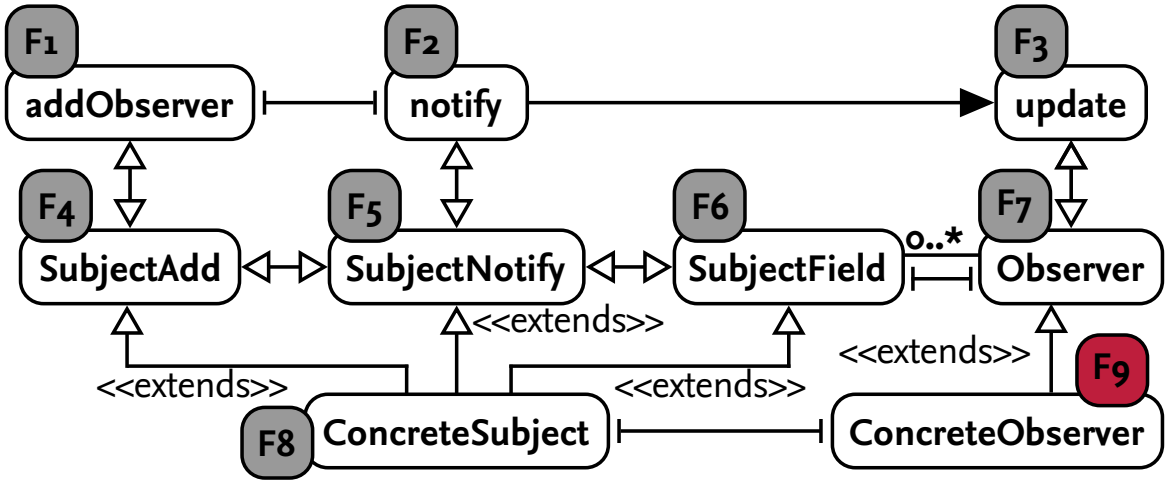
Family Role Model of Observer Pattern

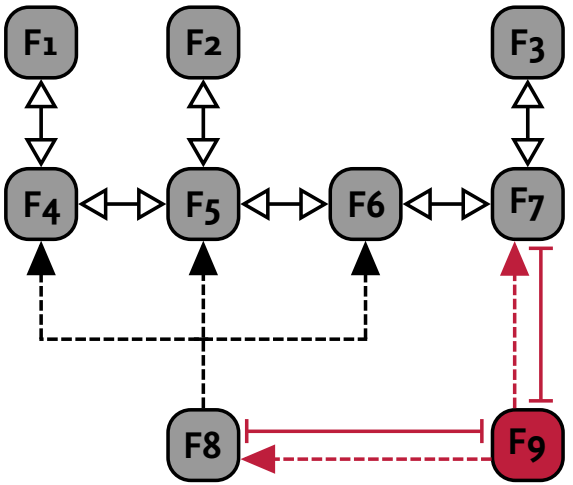


Family Role Model of Observer Pattern



Family Role Model of Observer Pattern





Part IV

Variability-Aware Design Pattern Catalog

Variability-Aware Design Pattern Catalog

Why?

- Variability-aware application of a pattern
- Describe application rules and guidelines

How?

- Based on design pattern catalog of GOF
- Intent, solution, consequences
- Using family role models

Decomposed Observer Pattern

Intent

Modularized event-notification

Solution

- *Subject* and *Observer*-related abstractions encapsulated
- *Concretes* scattered across feature model

Consequences

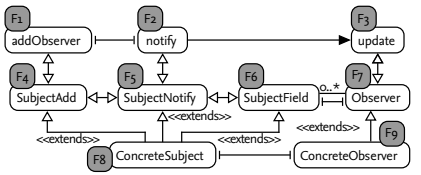
- Modularize subjects and observers by functionality
- Possible feature-optionality problem

Observer pattern

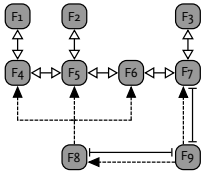
Intent

Create an event-notification of observing objects while modularizing concrete subjects and observers by functionality.

Solution



(a) Role model for the observer pattern annotated with features



(b) Role model for participating features of the observer pattern

We depict the structure of the solution above. In Figure 7.3a , we show the different roles with their introducing features, whereas in Figure 7.3b , we illustrate the relationships and dependencies of these features. We make the following suggestions:

- Introduce all abstracting Subject-related as well as Observer-related roles of Figure 7.3a together in one feature. The Subject is introduced using three roles, the SubjectAdd, SubjectNotify and SubjectField roles (feature roles F4 – F6 in Figure 7.3b). Hence, all three feature roles F4, F5 and F6 are equivalent. This also implies an equivalence to F1 and F2, introducing the addObserver and notify roels. Moreover, the Observer role with its update role should be introduced in the same features, implying an equivalence of F3 and F7 as well.
- Modularize different ConcreteSubjects (F8) by functionality, such that at least one is mandatory if the Subject (F4 – F6) exists.
- Modularize different ConcreteObservers (F9) by functionality. While requiring for a ConcreteSubject (F8) and the Observer (F7) to exist, no other relations to other features are necessary.

Consequences

Using this solution, the following consequences are implied:

+ The observer's functionality is encapsulated independently of subjects.
+ Modularization of abstractions and concrete implementations is allowed, easing the extensibility for new ConcreteSubjects and ConcreteObservers.
– The feature-optionality problem (cf. Section 2.2.3) is introduced if both, ConcreteSubjects and ConcreteObservers, are optional. Adding a ConcreteSubject might lead to unexpected updates for ConcreteObserers, which would have to be refined in order to understand new updates.

Part V

Conclusion & Future Work

Conclusion & Future Work

Contribution

- Variability-aware design patterns
  - Family role models
- Case study on common usage

Future Work

- Pattern-based SPL Design
  - Use Family Role Model
  - Generate variable realization artifacts