Modular Specification of Encapsulated Object-Oriented Components

Arnd Poetzsch-Heffter

Software Technology Group

University of Kaiserslautern

1. Introduction
2. Boxes: Encapsulated OO-Components
3. Modular Specification Technique
4. Conclusions

1. Introduction

1. Introduction: Specifications

Program specifations

- formulate properties of software units:
  - language-dependent (e.g. type-safety, no NullPointerException)
  - program-dependent (e.g. behavior of a particular method)
- general goals:
  - improve development, documentation, and understanding
  - support testing and dynamic checks
  - allow for verification (mathematical, static analysis, formal)
- software engineering goals:
  - seperation of interface and implementation
  - reuse and modularity

1. Introduction: Specifications and Components

Role and requirements of component specification

[Szyperski: Component Software, 2nd ed., p. 41 & p.78]

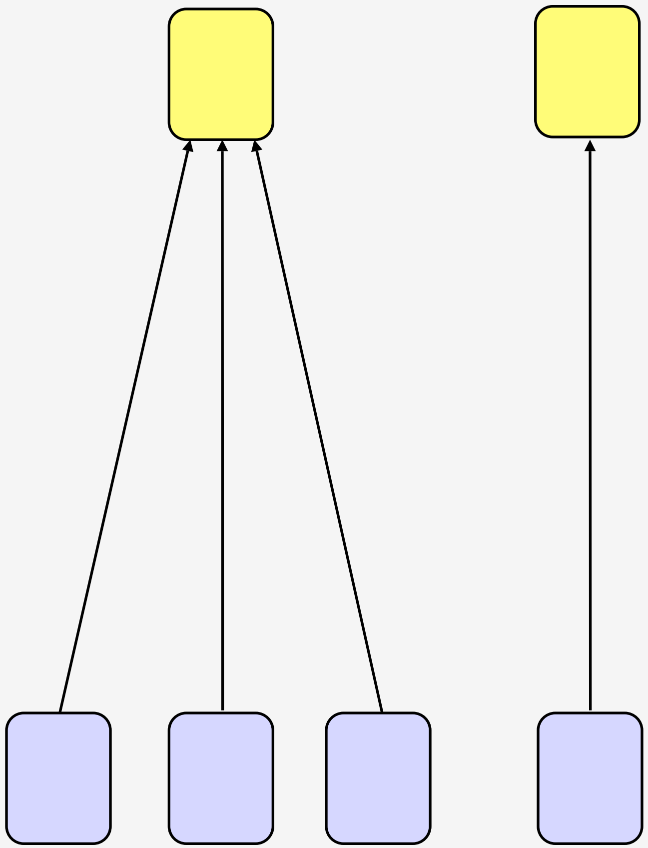
Definition of component:

"A software component is a unit of composition with contractually specificied interfaces and explicit context dependencies only. ..."

Discussion of component specification:

"The specification problems encountered in recursive re-entrant systems need to be solved in a modular way to cater for components. In other words, each component must be independently verifiable based on the contractual specification of the interfaces it requires and those it provides."

1. Introduction: Observable Game Example



```
box interface ObservableGame
{
   ObservableGame()
   void move( MoveDescr md )
   void swapPlayers()
   Position readPos()
   void register( Observer go )
}
```

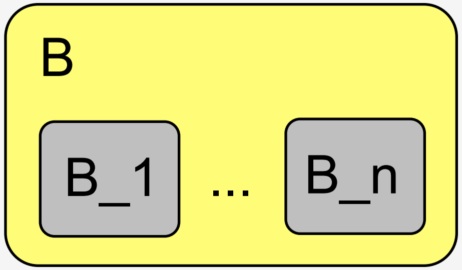
```
interface Observer
{
  void stateChanged()
}
```

```
box interface GameObserver
    implements Observer
{
   GameObserver( ObservableGame g )
   void stateChanged()
}
```

1. Introduction: Specification and Implementation

A <u>specified and implemented box class B</u> consists of:

- the specification S(B)
- subbox specifications S(B_1), ..., S(B_n)
- specifications for used interfaces and data S(E)
- an implementation I(B)
    - providing box-local functionality and
    - connecting the subboxes B_i



<u>Modular specification/verification:</u>
S(B_1), ... , S(B_n), S(E), I(B) |= S(B)

<u>Remarks:</u>

- Not solved for object-oriented programming
- Close relation between implementation and specification

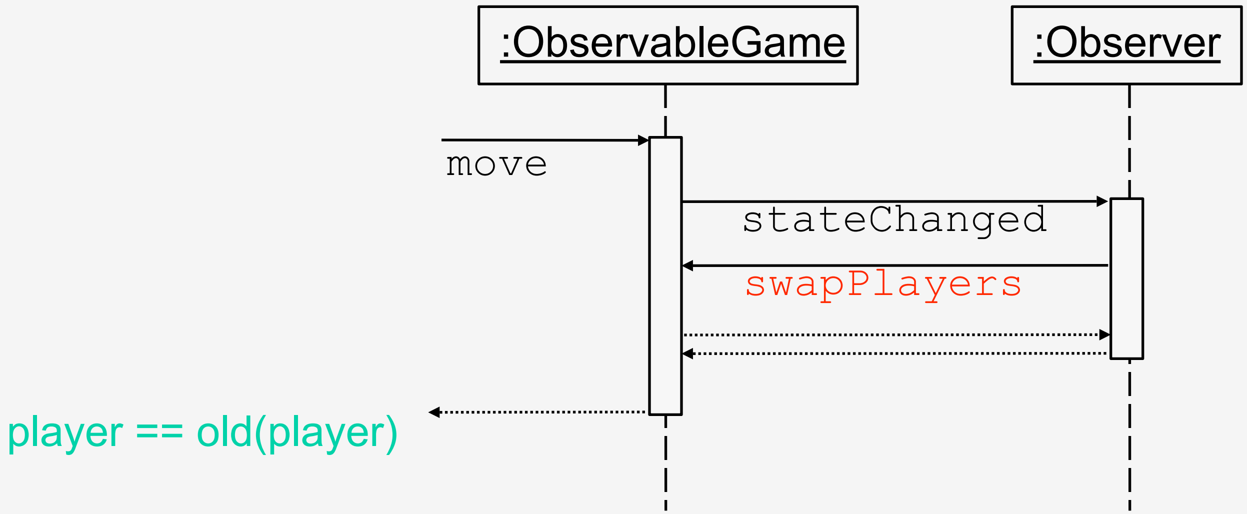
1. Introduction: Challenges of Modularity (1)

A method can affect its local state and its environment.

Locality principle:

Modular specifications have to be locally verifiable. In particular, they

- may only depend on the *local* state
- must control callbacks



1. Introduction: Challenges of Modularity (2)

Locality principle is not sufficient.

Frame principle:

Specification of method has to describe

- the effects on the environment
- the absence of effects on the environment

Otherwise specifications cannot be used in composition.

Example: Specification of method `move` has to state that `stateChanged` is invoked on the observer.

Problem:

Modularity implies that knowledge about environment is weak:

→ Effects have to be specified in an abstract way.

1. Introduction: Challenges of Modularity (3)

Software engineering requires more.

Composition principle:

Specifications have to be constructed from subbox specifications by

- providing access to subboxes
- abstraction and hiding of subboxes

This influences the component model.

Example: Gaming system encapsulating the registration mechanism:

```
box interface GamingSystem
{
   GamingSystem()
   Game createGame()
   SimpleGameObserver
          createSGameObserver( Game g )
}
```

```
interface Game
{
   void move(MoveDescr md)
   void swapPlayers()
}
```

```
interface SimpleGameObserver { }
```

1. Introduction: Overview of the Following

Structure of the following:

- Boxes: Encapsulated OO-components
- Modular specifications for boxes

→ Focus of talk is on the overall picture

2. Boxes: Encapsulated OO-Components

2. Boxes: Dynamic Encapsulation

Role of encapsulation and its boundaries:

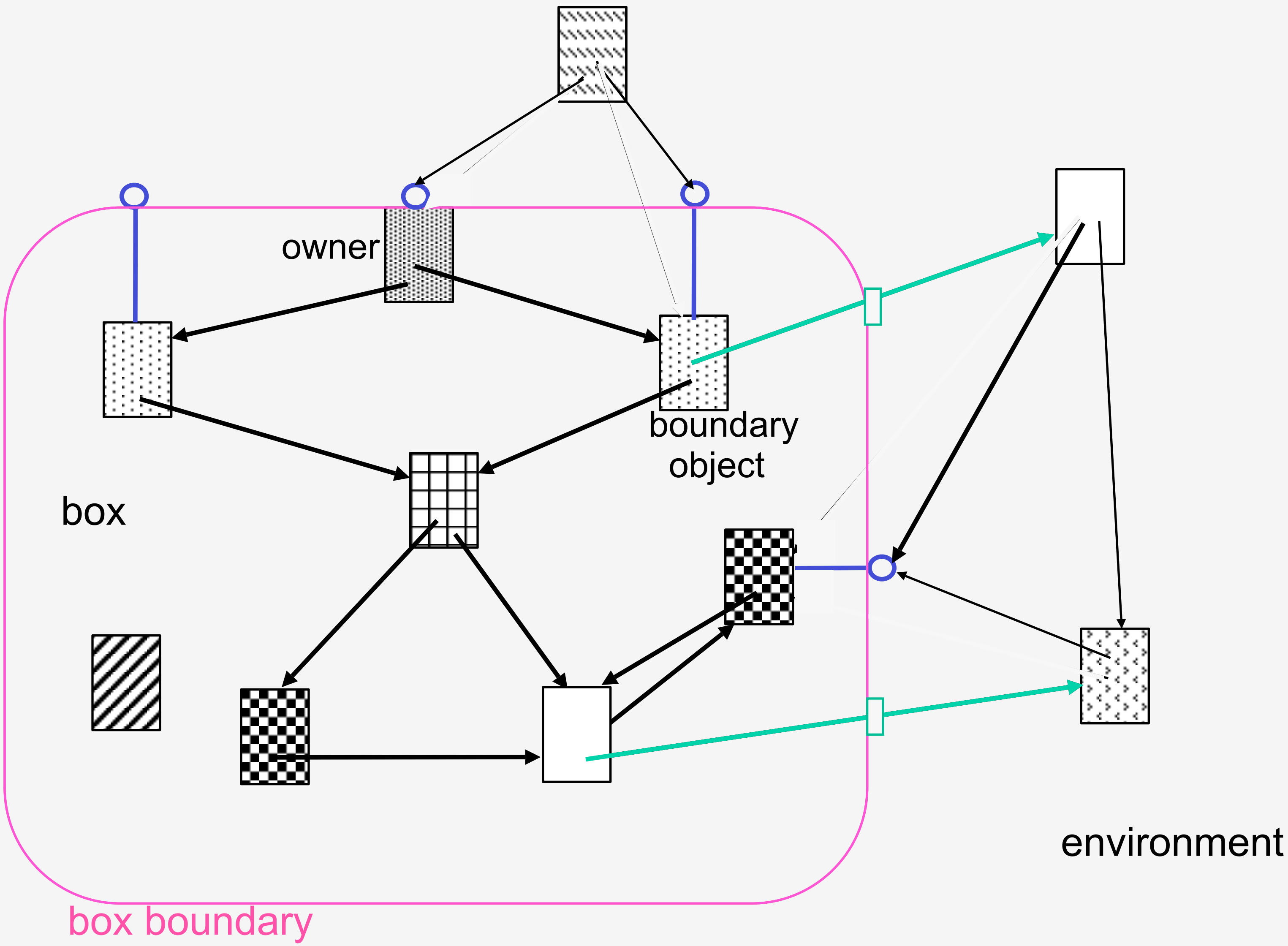
- structuring of the object space: local vs. non-local
- hiding and alias control
- provided interface and references to the outside
- unit of specification dependency
- (unit of locking and synchronization)

box = encapsulated set of objects + interface

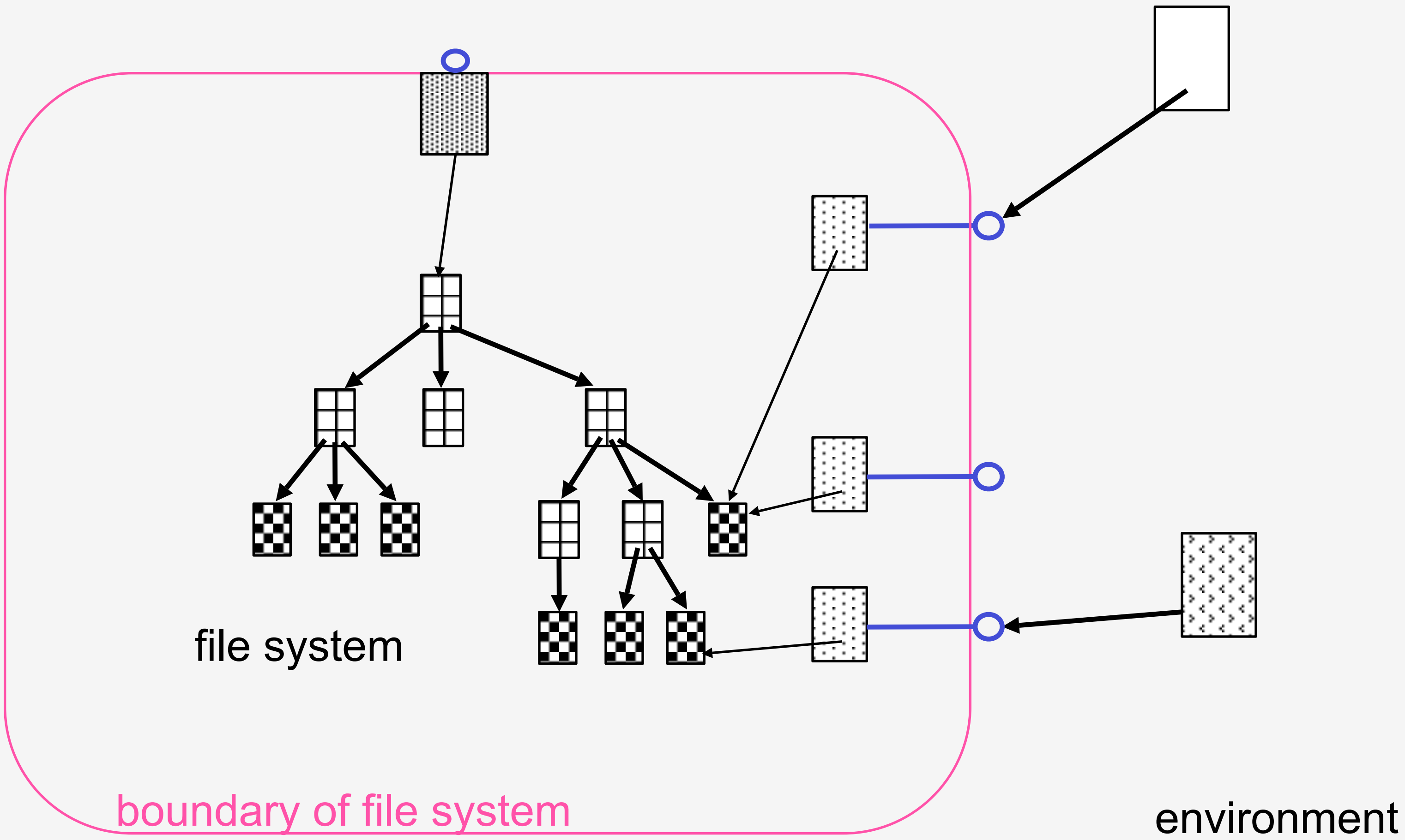
Relation to ownership techniques:

- ownership contexts with multiple ingoing references
- similar to ownership domains
- control of outgoing references

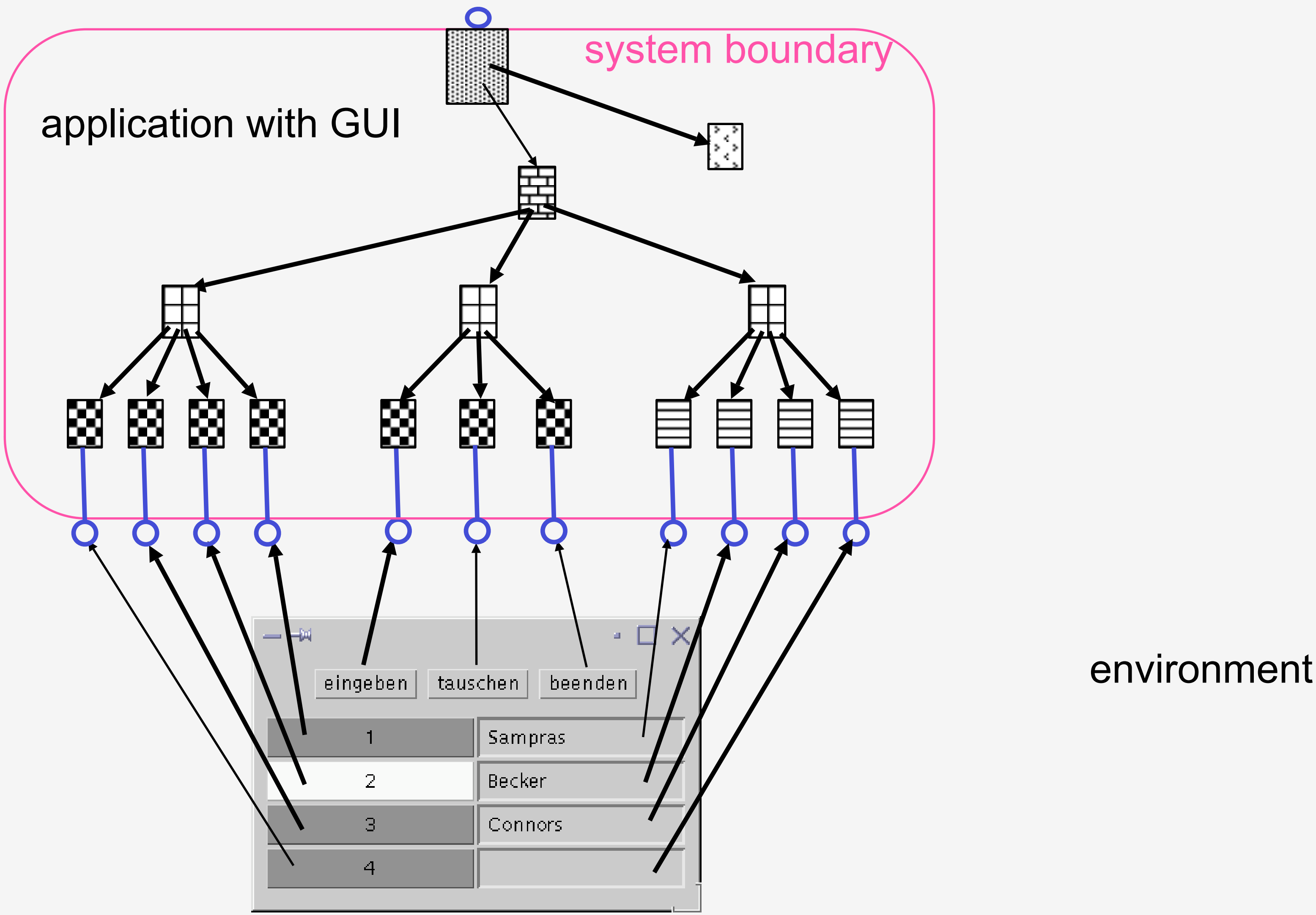
2. Boxes: Hierarchical Structured Object Systems



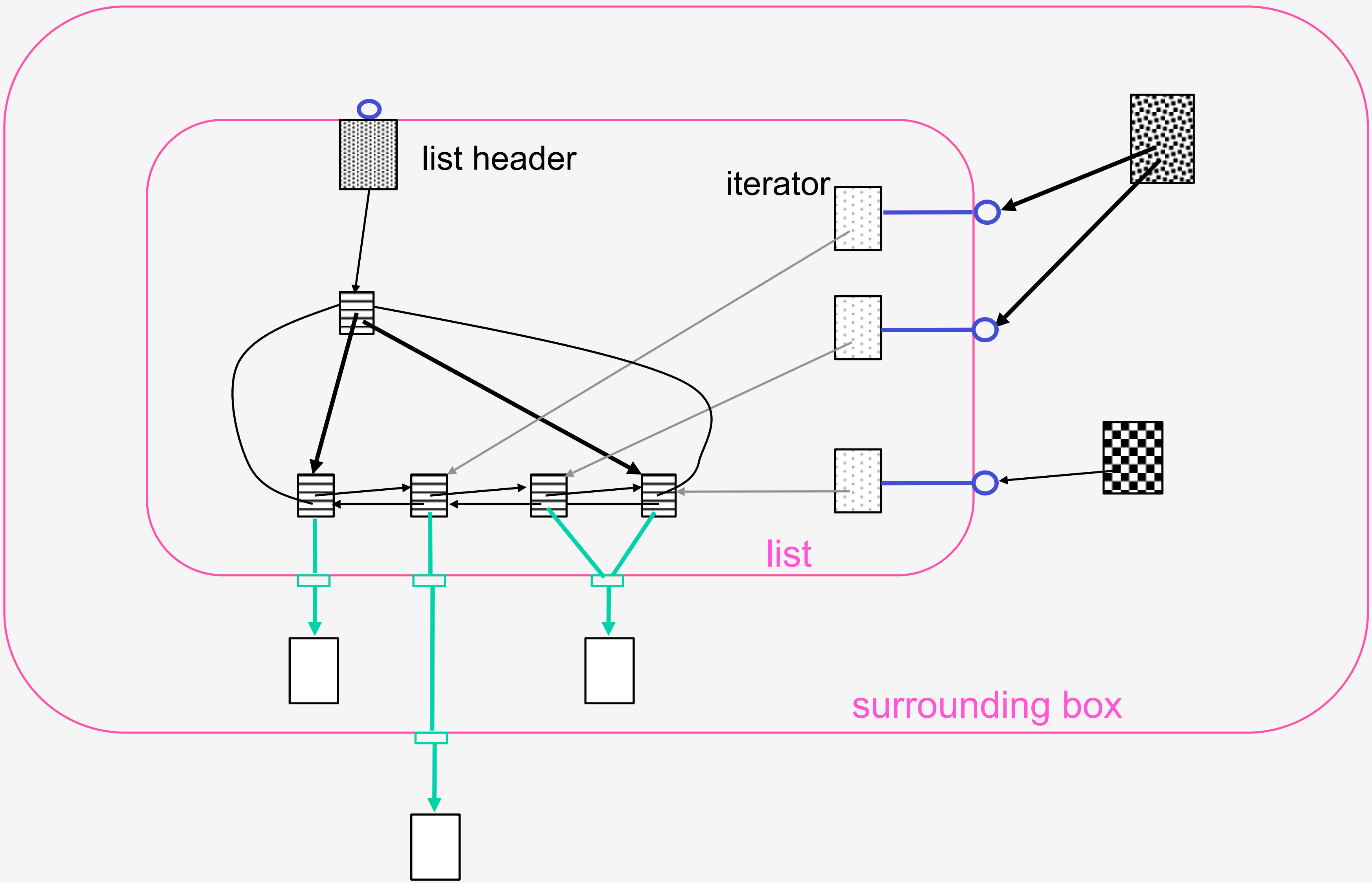
2. Boxes: Filesystem (Example 1)



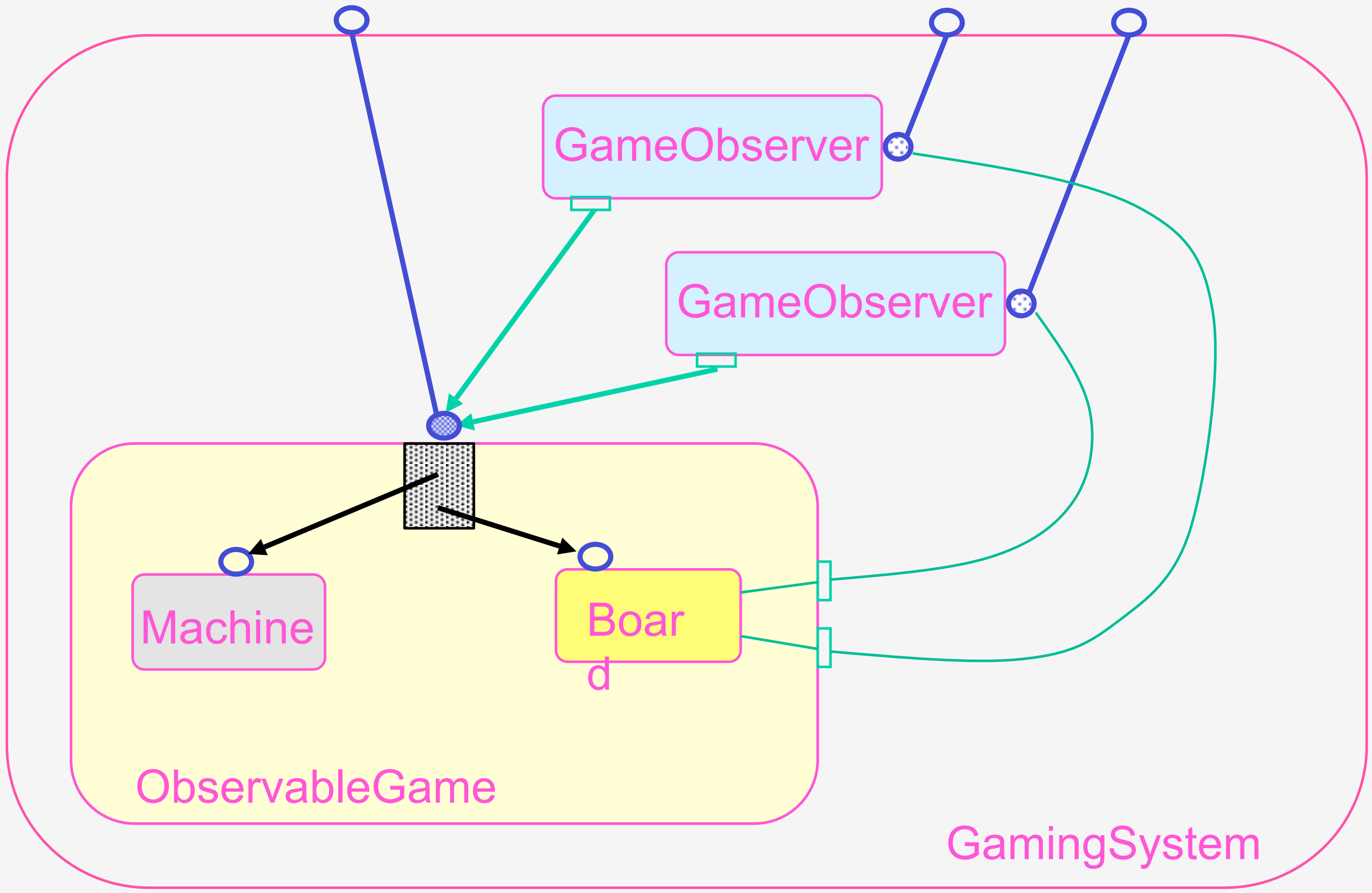
2. Boxes: Application with GUI (Example 2)



2. Boxes: Lists with Iterators (Example 4)



2. Boxes: Observable Games Example

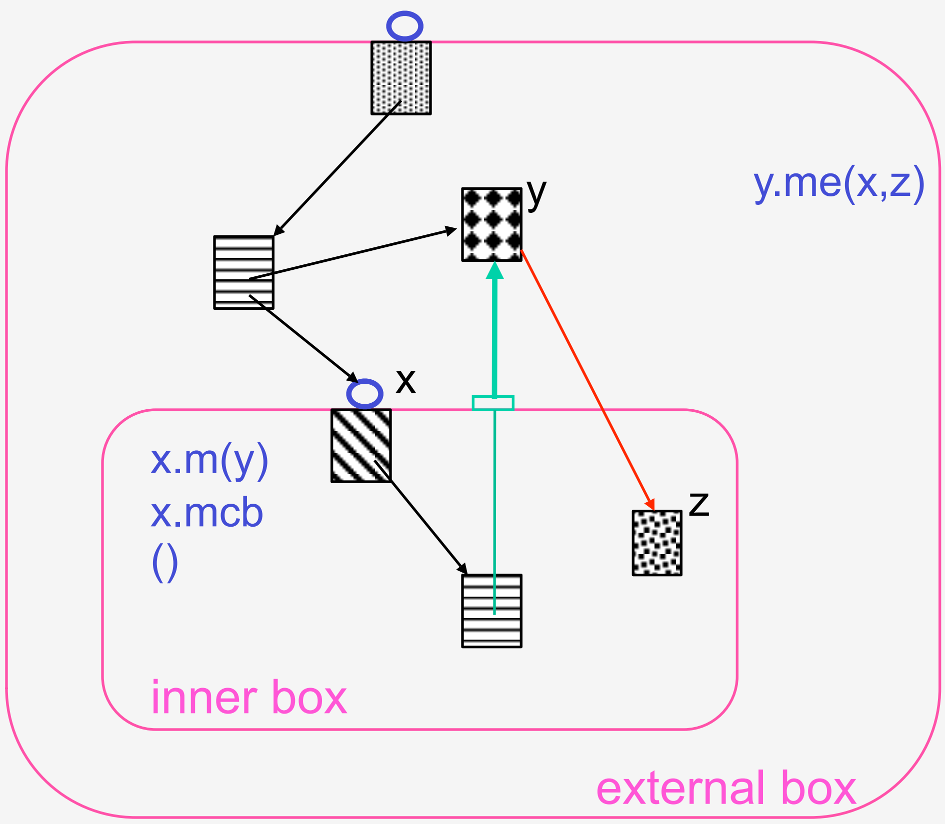


2. Boxes: Observations

Boxes are runtime instances that

- can have several objects of different classes at their boundary
- encapsulate objects of different classes
- can be implemented by modules, but not every module implements a box
- can change their „interface“ over time
- can be hierarchically structured
- provide *the* encapsulation boundaries

2. Boxes: Dynamic Behavior



Illustrating dynamics:

- object creation
- box creation
- local state change
- boundary call
- import of references
- outgoing call
- export of references
- callbacks

2. Boxes: More Details

<u>Further aspects of programming model:</u>

- only local object creation
- restrictions on down casts
- restrictions to enforce encapsulation

<u>Extensions:</u>

- membership transfer
- non terminating actions/methods
- asynchronous messages
- concurrency
- object deletion / live time restrictions on objects

3. Modular Specification Technique

3. Specification Technique: Overview

<u>Structure of specifications:</u>

- Specification and checking techniques for encapsulation
- Specification techniques for boxes:
  - state
  - invariants
  - method behavior:
    - local
    - frame
    - interaction / reentrance

3. Specification Technique: Encapsulation (1)

Box interface:

- provided interfaces
- referenced interfaces
- methods

Parameter constraints:

- without extension:
  - owner
  - boundary objects
  - pure
- with extension:
  - external objects

```
box interface  GamingSystem
{
  provides Game* games;
  provides SimpleGameObserver*
obs;

  GamingSystem()
  Game createGame()
  SimpleGameObserver
    createSGameObserver( Game g )
}
```

```
interface Game  {
  void move( MoveDescr md )
  void swapPlayers()
}
```

```
interface SimpleGameObserver { }
```

```
pure interface  MoveDescr  { ... }
```

3. Specification Technique: Encapsulation (2)

```
box interface ObservableGame
{
   references Observer* gameObs;
   ObservableGame()

    void move( MoveDescr md )
    void swapPlayers()
    void register( external Observer go )
    Position readPos()
}
```

```
interface  Observer {
    void stateChanged()
}
```

```
pure interface  MoveDescr  { ... }
```

```
pure interface  Position  { ... }
```

Difficulties:

- good notion of purity
- use of boundary objects as actual parameter (out-in)
- retrieving external objects (in-out)
- handling objects from different external boxes

3. Specification Technique: Encapsulation (3)

```
box interface  LinkedList<A>
{
   provides Iterator<A>*;
   references Object<A>*;
   LinkedList()

   external Object<A> get()
   void add( external Object<A> e )
   Iterator<A> listIterator() {
}
```

```
interface Iterator<C> {
  boolean hasNext();
  external Object<C> next();
}
```

Checking approach:

- encapsulation type system similar to ownership types/domains
- type inference and checking

3. Specification Technique: Box state

Box state is specified by the

- concrete
- abstract (model/ghost)

fields (private, spec public)

of the owner and the boundary objects.

Abstract fields may only depend on the fields of the box.

Example:

```
interface Game
{
    Position currentPos;
    Color  player;
    void move( MoveDescr md )

    void swapPlayers()
}
```

```
interface SimpleGameObserver {
    Position observedPos;
    Game obsGame;
}
```

3. Specification Technique: Invariants (1)

Specification invariant for GamingSystem:

```
invariant
  forall o in obs:  o.observedPos == o.obsGame.currentPos
```

Problems with invariant:

- Where should they hold?
- What are the fields they may depend on?
- Invariants cause a modularity problem:

Invariants of *all* classes have to hold in the prestate of a call!

Example:

```
class  C  {
   void  me( D x ) {
       ...
       x.foo();
} }
```

3. Specification Technique: Invariants (2)

Approach to invariants:

- Invariants may only depend on the fields of the box.
- Invariants have to hold whenever the thread is outside the box.
- This helps to solve the modularity problem.

Discussion:

- Implicit unpack/pack mechanism whenever box boundary is crossed.
- Invariant may depend on execution state (*type states*).

3. Specification Technique: Method Behavior (1)

Method specification:

- Changes to local state and result
- Changes to the environment
- What is left unchanged
- Reentrance behavior

Frame problem

Example:

```
interface  Observer {
    void stateChanged()
}
```

```
box interface ObservableGame
{
  references Observer* gameObs;
  ObservableGame()

   void move( MoveDescr md )
   void register( external Observer go )
   ...
}
```

3. Specification Technique: Method Behavior (2)

Existing approach to frame problem:

- Describe what is allowed to be modified (modifies clause)
- What is not mentioned in the modifies clause may not change
- Loose coupling, information hiding, and abstraction is difficult to handle

Box-based approach to frame problem:

- Specify what is left unchanged in the box
- Specify calls on external objects

allows for modular verification

3. Specification Technique: Method Behavior (3)

Technique for specifying outgoing calls:

- Refinement calculi / grey box specifications (R. Back / M. Büchi)
- (Process calculi)

Example:

```
box interface ObservableGame
{  ...
   void move( MoveDescr md )
     requires  legal(md,currentPos)
     behavior
        currentPos = doMove(md,currentPos);
        forall o in gameObs { o.stateChanged()  }
        any( cmd :: legal(cmd,currentPos) ) {
           currentPos = doMove(md,currentPos);
           forall o in gameObs { o.stateChanged() }
        }
      ensures  unchanged([player,state,gameObs])
... }
```

Problem:

Reentrance

3. Specification Technique: Method Behavior (4)

Approach to reentrance:

- Grey box specifications
- Type states to restrict

  callable methods

Example:

Only readPos is executable
if
state == OBSERVABLE

```
void move( MoveDescr md )
    requires      legal(md,currentPos)
                  && state == VALID
    behavior
        state = OBSERVABLE;
        currentPos = doMove(md,currentPos);
        forall o in gameObs {  o.stateChanged()  }
        any( cmd :: legal(cmd,currentPos) ) {
            currentPos = doMove(md,currentPos);
            forall o in gameObs {o.stateChanged() }
        }
        state = VALID;
    }
    ensures  unchanged([player,state,gameObs])
```

4. Conclusions

4. Conclusions

Summary:

- Structuring techniques for object stores → boxes
- Enforcing encapsulation
- Specification techniques for boxes

Conclusions:

- Encapsulation with semantical guarantees is central for modularity.
- Programming models provide a good basis for component models.
- Interface specifications and programming languages cannot live in different worlds.
- Some architectural elements might be helpful for programming.

4. Conclusions: ...

Current and future work:

- Finishing the encapsulation system
- Concurrency models based on boxes
- Realizing a lightweight specification support for a Java subset
- Verification techniques for the approach
- Substitutability: “Box subtyping”
- Examples, examples, ...

Questions?