

# Concepts of Object-Oriented Programming

**Peter Müller**

Chair of Programming Methodology

Autumn Semester 2009



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Visible States

- Invariants have to **hold in pre- and poststates** of methods executions but may be **violated temporarily** in between
- Pre- and poststates are called “**visible states**”

```
class Redundant {  
    private int a, b;  
    // invariant a == b  
  
    public void set( int v ) {  
        // invariant of this holds  
        a = v;  
        // invariant of this violated  
        b = v;  
        // invariant of this holds  
    }  
}
```

# 9. Object Invariants

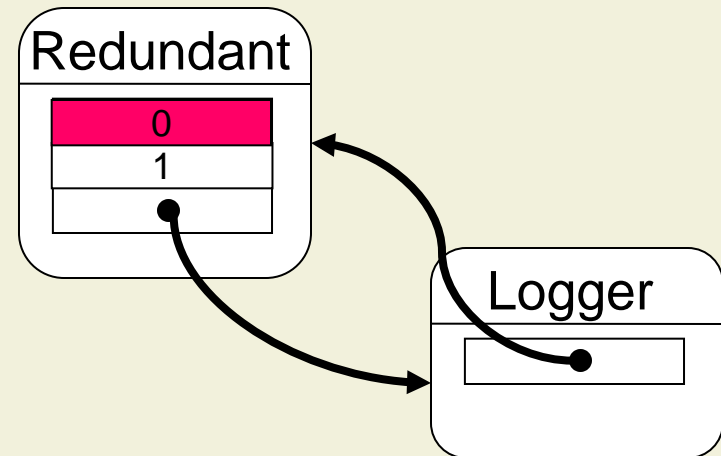
## 9.1 Call-backs

## 9.2 Invariants of Object Structures

# Call-backs

```
class Redundant {  
    private int a, b;  
    private Logger l;  
    // invariant a == b  
    public void set( int v ) {  
        a = v;  
        l.log( "Inside set" );  
        b = v;  
    }  
  
    public int div( int v ) {  
        return v / ( a - b + 1 );  
    }  
}
```

```
class Logger {  
    private Redundant r;  
  
    public void log( String m ) {  
        System.out.println( m + r.div( 5 ) );  
    }  
}
```



# Common Variations

## ▪ Self-calls

```
class Redundant {  
    private int a, b;  
  
    // invariant a == b  
  
    public void set( int v ) {  
        a = v; this.div( 5 ); b = v;  
    }  
  
    public int div( int v ) {  
        return v / ( a - b + 1 );  
    }  
}
```

## ▪ Re-entrant monitors

```
class Redundant {  
    private int a, b;  
  
    // monitor invariant a == b  
  
    public synchronized void set( int v ) {  
        a = v; this.div( 5 ); b = v;  
    }  
  
    public synchronized int div( int v ) {  
        return v / ( a - b + 1 );  
    }  
}
```

Java

# Running Example

```
class Account {  
    int balance;  
    Currency! cur;  
    Regulator! regulator;  
  
    // invariant  cur == Currency.CHF ==> balance % 5 == 0;  
  
    void Exchange( Currency! c ) {  
        balance = cur.Convert( balance, c );  
        cur = c;  
        regulator.Report( this );  
        if( cur == Currency.CHF )  
            balance = balance / 5 * 5;  
    }  
    ...  
}
```

# Solution 1: Re-establishing Invariants

- Check invariant before every method call
- Overly restrictive: most methods do not call back
- Too expensive for run-time checking

```
class Account {  
    int balance;  
    Currency! cur;  
    Regulator! regulator;  
  
    // invariant  cur == Currency.CHF  
    //              ==> balance % 5 == 0;  
  
    void Exchange( Currency! c ) {  
        balance = cur.Convert( balance, c );  
        cur = c;  
        if( cur.Equals( Currency.CHF ) )  
            balance = Round( balance );  
        regulator.Report( this );  
    }  
    ... }
```

# Solution 2: Call-back Analysis

- **Statically analyze code** of callee to detect call-backs
  - Check invariant before call **only if call-back is possible**
- **Not modular**
  - For dynamically-bound methods, all overrides need to be known

```
class Account {  
    int balance;  
    Currency! cur;  
    Regulator! regulator;  
  
    // invariant  cur == Currency.CHF  
    //              ==> balance % 5 == 0;  
  
    void Exchange( Currency! c ) {  
        balance = cur.Convert( balance, c );  
        cur = c;  
        regulator.Report( this );  
        if( cur == Currency.CHF )  
            balance = balance / 5 * 5;  
    }  
    ... }
```

What if  
Regulator is  
an interface?



# Solution 3: Explicit Requirements

- Specify in each precondition which invariants the method actually requires
- Check required invariants before method call

```
class Account {  
    ...  
    // requires invariant of this and c;  
    void Exchange( Currency! c ) {  
        balance = cur.Convert( balance, c );  
        cur = c;  
        regulator.Report( this );  
        if( cur == Currency.CHF )  
            balance = balance / 5 * 5;  
    }  
    ...  
}
```

# Explicit Requirements: Problems

- Writing the concrete invariant in precondition **violates information hiding**
- Some methods require a **large number** of invariants
  - For example, tree traversal

```
class Account {  
    ...  
    // requires invariant of this and c;  
    void Exchange( Currency! c ) {  
        balance = cur.Convert( balance, c );  
        cur = c;  
        regulator.Report( this );  
        if( cur == Currency.CHF )  
            balance = balance / 5 * 5;  
    }  
    ...  
}
```

## Solution 4: Dented Invariants

- Use boolean field to indicate whether object is **valid** or not
  - Can be used to turn invariant on and off
- Dented invariant holds **in all visible states**
- Explicit requirements can be stated using the valid-field

```
class Account {  
    ...  
    boolean valid;  
  
    // invariant  valid ==>  
    //             cur == Currency.CHF  
    //             ==> balance % 5 == 0;  
  
    // requires this.valid && c.valid;  
    void Exchange( Currency! c )  
    { ... }  
    ...  
}
```

# Re-establishing Dented Invariants

- Programmers might forget to set valid-field
- Invariants still **need to be checked before method calls**
- A method can break many invariants through direct field updates

```
class Account {  
    boolean valid;  
  
    void Exchange( Currency! c ) {  
        balance = cur.Convert( balance, c );  
        cur = c;  
        regulator.Report( this ),  
        if( cur == Currency.CHF )  
            balance = balance / 5 * 5;  
    }  
    ...  
}
```

valid not  
set to false

Dented  
invariant  
does not  
hold

# Basic Spec# Methodology

- Each object has an implicit valid-field
  - **Valid** and **mutable** objects
- Each invariant is **implicitly dented**
- Object invariants can depend **only on the fields of the *this* object** (will be relaxed later)
- Enforce that dented invariants hold **in all execution states**, not just visible states
  - **Un-dented invariant holds whenever an object is valid**
- **Valid objects must not be modified**
  - Check for each field update  $o.f = e$  that  $o$  is mutable

# Spec# Methodology: Example

```
class Account {
```

```
...
```

```
invariant cur == Currency.CHF ==> balance % 5 == 0;
```

```
void Exchange( Currency! c )
```

```
// requires this.valid && c.valid;
```

```
{
```

```
    balance = cur.Convert( balance, c );
```

```
    cur = c;
```

```
    regulator.Report( this );
```

```
    if( cur == Currency.CHF )
```

```
        balance = balance / 5 * 5;
```

```
    }
```

```
...
```

```
}
```

Invariant is  
implicitly dented

Check fails:  
receiver is  
not mutable

Implicit precondition:  
arguments are valid

Spec#

# Maintaining Object Validity

- Setting the valid-field to true might break the dented invariant
- valid-field can be **modified only through special `expose` block statement**
  - Exposed object must be initially valid
  - Similar to non-reentrant lock-block

Invariant violation goes undetected

```
int f;  
invariant 0 < f;  
void foo( ) {  
    valid = false;  
    f = -1;  
    valid = true;  
}
```

Set valid to false

Check invariant

Set valid to true

```
int f;  
invariant 0 < f;  
void foo( ) {  
    expose( this ) {  
        f = -1;  
    }  
}
```

Spec#

# Example Revisited

```
class Account {  
    invariant cur == Currency.CHF ==> balance % 5 == 0;  
  
    void Exchange( Currency! c )  
        // requires this.valid && c.valid;  
    {  
        expose( this ) {  
            balance = cur.Convert( balance, c );  
            cur = c;  
            regulator.Report( this );  
            if( cur == Currency.CHF )  
                balance = balance / 5 * 5;  
        }  
    }  
    ... }  
}
```

Check  
succeeds:  
**this** is valid

Check  
succeeds:  
receiver is  
mutable

Check  
succeeds:  
invariant holds

Spec#



# Establishing Object Validity

- New objects are **initially mutable**
  - valid-Field is initialized to false
- After initialization, un-dented **invariant** is checked and valid-field is set to true
  - We ignore inheritance here

```
class Account {  
    ...  
    invariant cur == Currency.CHF  
                ==> balance % 5 == 0;  
  
    Account( Regulator! r ) {  
        cur = Currency.CHF;  
        regulator = r;  
    }  
    ...  
}
```

Invariant holds since balance == 0

Implicit:  
**this.valid = true;**

Spec#

# Basic Spec# Methodology: Summary

- **Admissible invariants**

- The invariant of an object *o* may depend on fields of *o* (and constants)

- **Checks** (proof obligations)

- Invariant of *o* holds after *o* has been initialized
- Invariant of *o* holds at the end of each **expose**( *o* ) block
- Every expose operation is done on a valid object
- Every field update is done on a mutable receiver

- Recall: we ignore inheritance here

# Call-backs in Spec#: Example

```
class Account {  
  void Exchange( Currency! c )  
    // requires this.valid && c.valid;  
  {  
    expose( this ) {  
      ...  
      regulator.Report( this );  
      ...  
    }  
  }  
  
  int GetBalance( )  
    // requires this.valid  
  { return balance; }  
}
```

Spec#

In principle, methods  
can be called while  
invariant is broken

This call is forbidden  
since precondition  
does not hold

Requirement about  
expected invariants

```
class Regulator {  
  void Report( Account! a  
    // requires this.valid && a.valid;  
  {  
    int b = a.GetBalance( );  
    // ...  
  }  
  ...  
}
```

Spec#

# Call-backs in Spec#: Example (cont'd)

```
class Account {  
  void Exchange( Currency! c )  
    // requires this.valid && c.valid;  
  {  
    expose( this ) {  
      ...  
      regulator.Report( this );  
      ...  
    }  
  }  
  
  int GetBalance( )  
    // requires this.valid  
  { return balance; }  
}
```

Call is allowed since  
precondition holds

Spec#

```
class Regulator {  
  void Report( Account! a )  
    // requires this.valid;  
  {  
    int b = a.GetBalance( );  
    // ...  
  }  
  ...  
}
```

a's invariant is  
not expected

Call-back is forbidden  
since precondition  
does not hold

Spec#

# Call-backs in Spec#: Example (cont'd)

```
class Account {  
  void Exchange( Currency! c )  
    // requires this.valid && c.valid;  
  {  
    expose( this ) {  
      ...  
      regulator.Report( this );  
      ...  
    }  
  }  
  
  int GetBalance( )  
    // requires true;  
  { return balance; }  
}
```

Call is allowed since  
precondition holds

No invariant  
expected

Spec#

```
class Regulator {  
  void Report( Account! a )  
    // requires this.valid;  
  {  
    int b = a.GetBalance( );  
    // ...  
  }  
  ...  
}
```

a's invariant  
not expected

Call-back is allowed  
since precondition  
holds

Spec#

# 9. Object Invariants

## 9.1 Call-backs

## 9.2 Invariants of Object Structures

# Multi-Object Invariants: Example

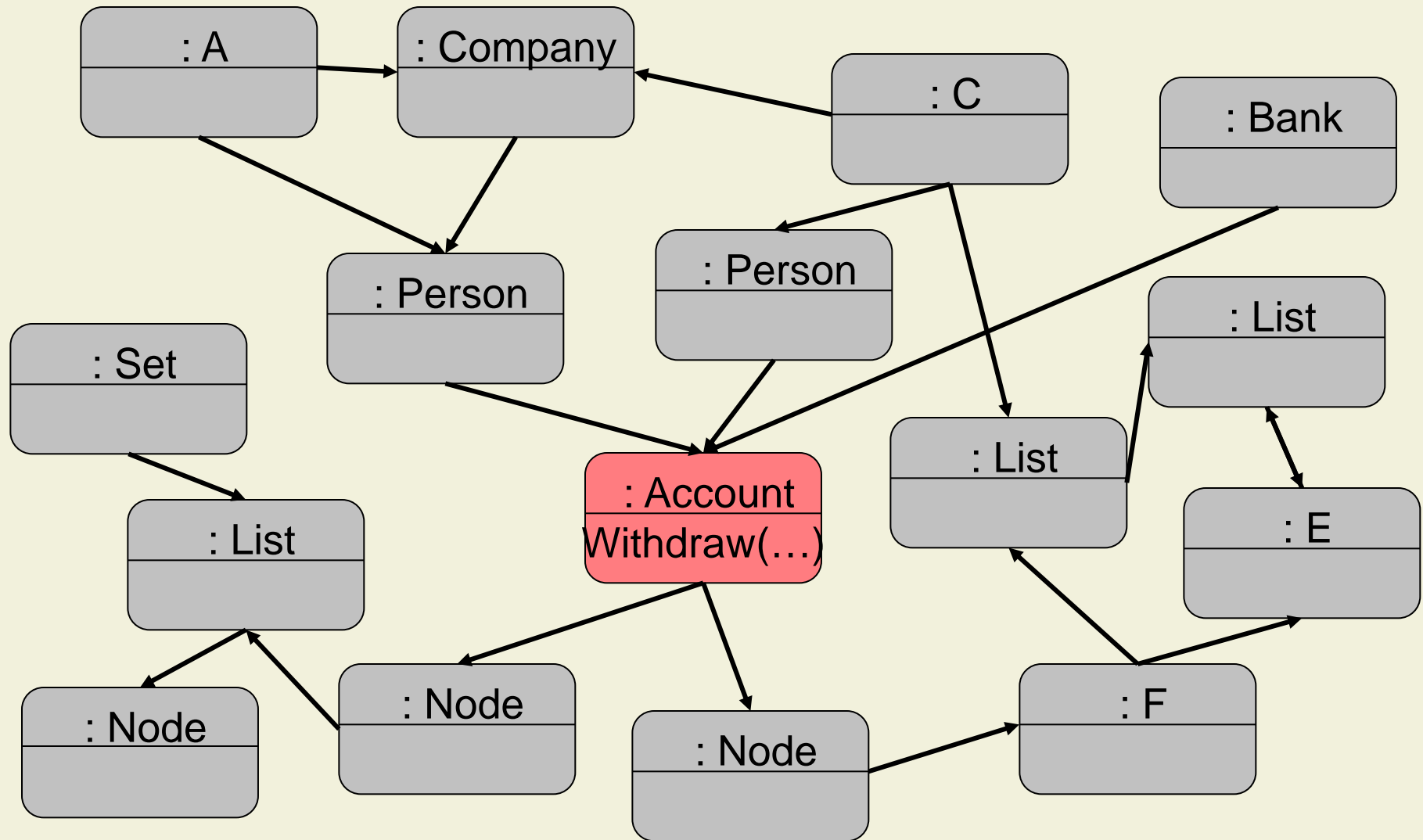
```
class Account {  
  ...  
  
  void Withdraw( int amount )  
    requires cur == Currency.CHF  
      ==> amount % 5 == 0;  
  ensures balance ==  
    old( balance ) – amount;  
{  
  expose( this ) {  
    balance = balance – amount;  
  }  
}  
}
```

Field update might  
break invariants of  
client objects

```
class Person {  
  Account! savings;  
  
  invariant 0 <= savings.balance;  
  ...  
}
```

Invariant  
depends on field  
of another object

# Finding Dependent Objects





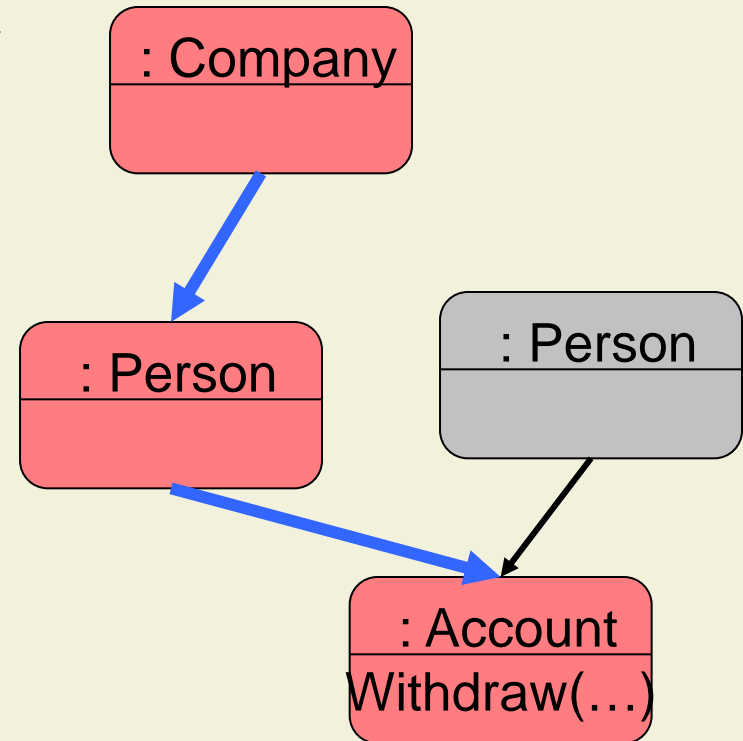
# Ownership-Based Invariants

## ■ Admissible invariants

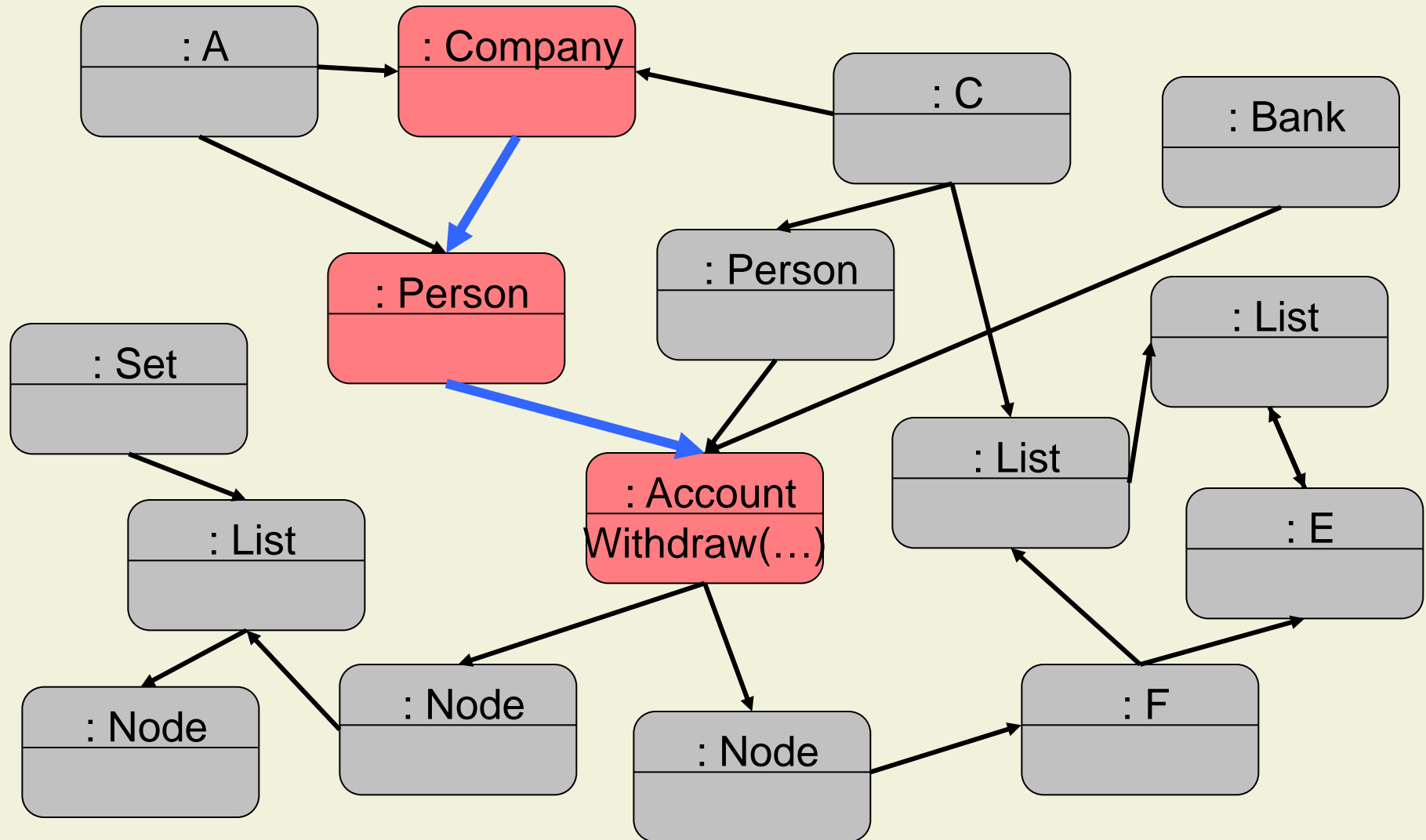
- The invariant of an object *o* may depend on fields of *o* **and objects (transitively) owned by *o*** (and constants)

## ■ Requirement: when an object *o* is mutable, so are *o*'s (transitive) owners

- Because an update of *o* might break the owners' invariants



# Using Ownership to Limit Dependencies



# Admissible Ownership-Based Invariants

```
class Person {  
  Account! savings;  
  
  invariant 0 <= savings.balance;  
  
  ...  
}
```

Use topological  
type system

Not admissible: invariant  
depends on field of  
another object that is not  
owned by **this**

```
class Person {  
  rep Account! savings;  
  
  invariant 0 <= savings.balance;  
  
  ...  
}
```

Spec# syntax:  
[Rep]

Admissible: savings  
is owned by **this**

# Mutable Owners: Example

```
class Account {  
  ...  
  
  void Withdraw( int amount )  
    requires cur == Currency.CHF  
      ==> amount % 5 == 0;  
    ensures balance ==  
      old( balance ) – amount;  
  {  
    expose( this ) {  
      balance = balance – amount;  
    }  
  }  
}
```

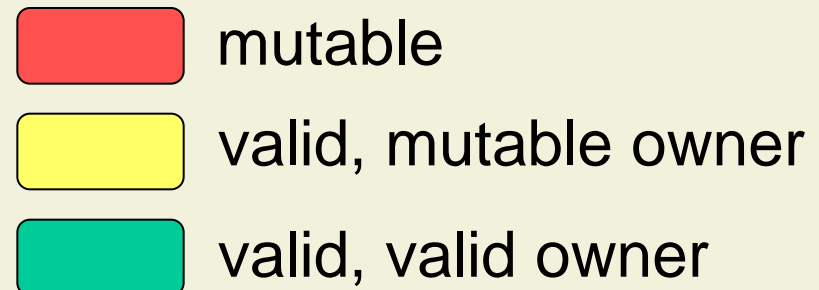
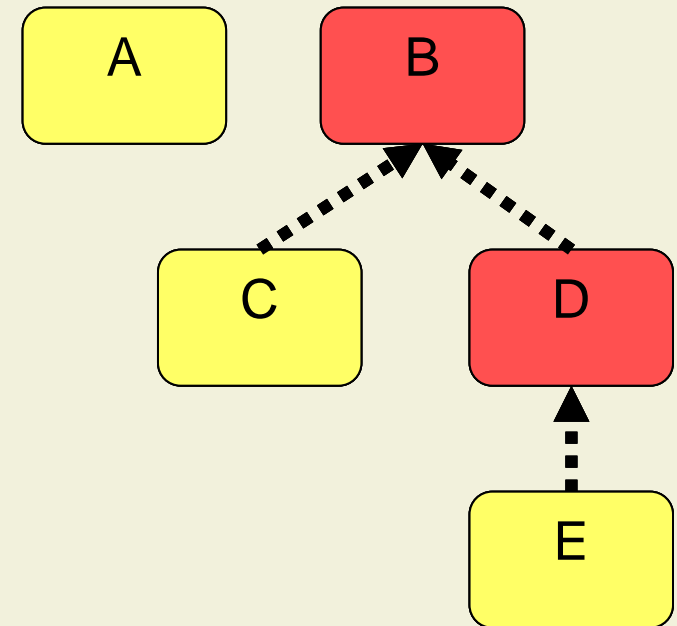
```
class Person {  
  rep Account! savings;  
  invariant 0 <= savings.balance;  
  void Donate( )  
  {  
    savings.Withdraw( 1000 );  
  }  
}
```

This call might  
break the  
invariant of **this**

Invariant of **this** is  
not checked!

# Enforcing Mutable Owners

- Rules
  - Expose owner before owned object
  - Un-expose in reverse order
- Additional checks for **expose( o )**
  - Before expose, o must be valid and o's owner must be mutable
  - At the end of expose, all objects owned by o must be valid



# Mutable Owners: Example (cont'd)

```
class Account {  
  ...  
  
  void Withdraw( int amount )  
    // requires valid && !owner.valid  
    requires cur == Currency.CHF  
      ==> amount % 5 == 0;  
    ensures balance ==  
      old( balance ) – amount;  
  {  
    expose( this ) {  
      balance = balance – amount;  
    }  
  }  
}
```

```
class Person {  
  rep Account! savings;  
  
  invariant 0 <= savings.balance;  
  
  void Donate( )  
    // requires valid && !owner.valid  
  {  
    savings.Withdraw( 1000 );  
  }  
}
```

This call is forbidden  
since precondition  
does not hold

# Mutable Owners: Example (cont'd)

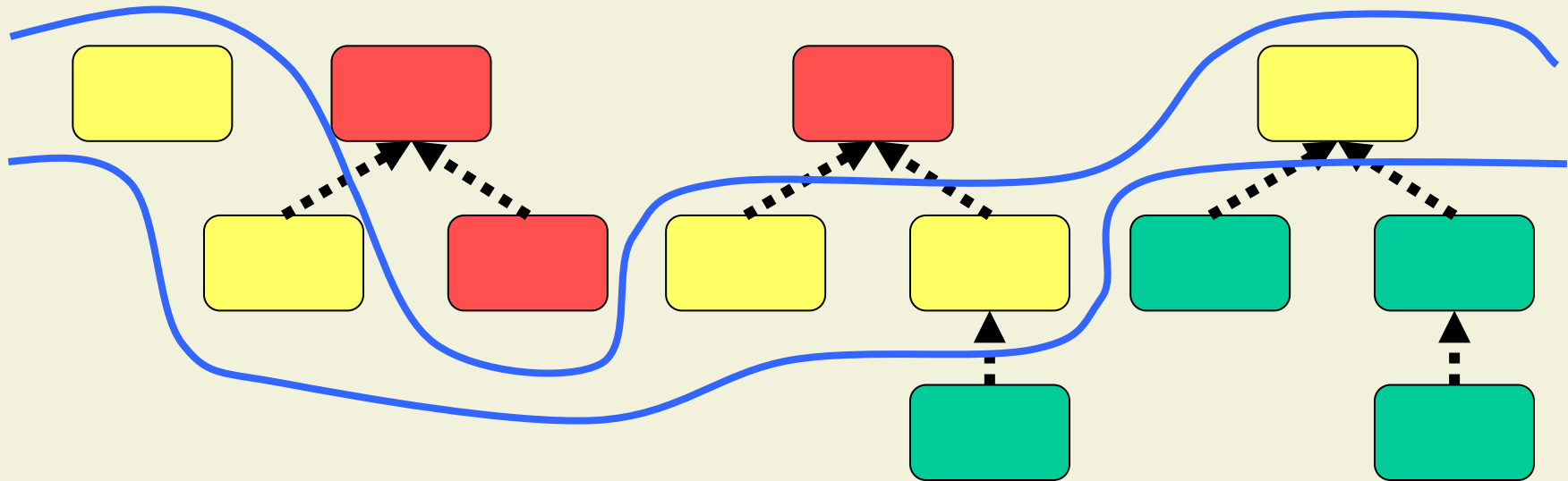
```
class Account {  
  ...  
  
  void Withdraw( int amount )  
    // requires valid && !owner.valid  
    requires cur == Currency.CHF  
      ==> amount % 5 == 0;  
    ensures balance ==  
      old( balance ) – amount;  
{  
  expose( this ) {  
    balance = balance – amount;  
  }  
}
```


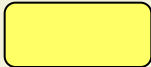

```
class Person {  
  rep Account! savings;  
  
  invariant 0 <= savings.balance;  
  
  void Donate( )  
    // requires valid && !owner.valid  
{  
    expose( this ) {  
      savings.Withdraw( 1000 );  
    }  
}
```

Call is allowed since  
precondition holds

Invariant check fails  
(add precondition to fix)

# Heap Snapshot



-  mutable
-  valid, mutable owner
-  valid, valid owner



# Spec# Methodology: Summary

- Admissible invariants
  - The invariant of an object  $o$  may depend on fields of  $o$  and objects (transitively) owned by  $o$  (and constants)
- Checks (proof obligations)
  - Owner of newly created object is mutable
  - Invariant of  $o$  holds after  $o$  has been initialized
  - Invariant of  $o$  holds at the end of each **expose**(  $o$  ) block and all objects owned by  $o$  are valid
  - Every expose operation is done on a valid object with a mutable (or no) owner
  - Every field update is done on a mutable receiver

# Spec# Methodology: Observations

- Methodology relies on **encapsulation** of object structures
  - No strict enforcement of owner-as-modifier discipline
  - But: owner must be exposed before owned object
- Responsibility for invariant checking is divided
  - A method **implementation** is responsible for the objects in the context of the receiver
  - A **caller** is responsible for the objects in its context
- Ownership-based invariants are **too restrictive** for many useful examples

# Invariants and Immutability

- Immutable objects can be **freely shared**
- Invariants may depend on the state of shared immutable objects
- Immutability often leads to **more reliable programs**
  - Especially for concurrency
  - If performance permits

```
[Immutable] class Integer {  
    int value;  
    ...  
}
```

Spec#

```
class Client {  
    Integer! i;  
    invariant 0 < i.value;  
    ...  
}
```

Spec#

No ownership necessary

# Invariants and Monotonicity

- Many properties of objects evolve monotonically
  - Numbers grow or shrink monotonically
  - Reference go from null to non-null
- Invariants may depend on properties of shared objects guaranteed by their history constraint

```
class Counter {  
  int value;  
  // constraint old( value ) <= value;  
  ...  
}
```

```
class Client {  
  Counter! c;  
  invariant 0 < c.value;  
  ...  
}
```

No ownership  
necessary

# Invariants and Visibility

- Invariants may depend on fields of shared objects if a **modular static analysis** can determine all necessary checks
- Invariant and field are declared in the same module
  - Common example:  
**recursive data structures**

```
class Person {  
  Person spouse;  
  invariant spouse == null ||  
    spouse.spouse == this;  
  ...  
}
```

No ownership necessary

Spec#

# Summary

- Sound, modular checking of object invariants is **surprisingly difficult**
  - Call-backs
  - Multi-object invariants
  - Inheritance
- Spec# is the first system to support sound, modular verification of object invariants
  - Efficient run-time checking does not seem feasible
- Spec# is **open source**: [specsharp.codeplex.com](http://specsharp.codeplex.com)