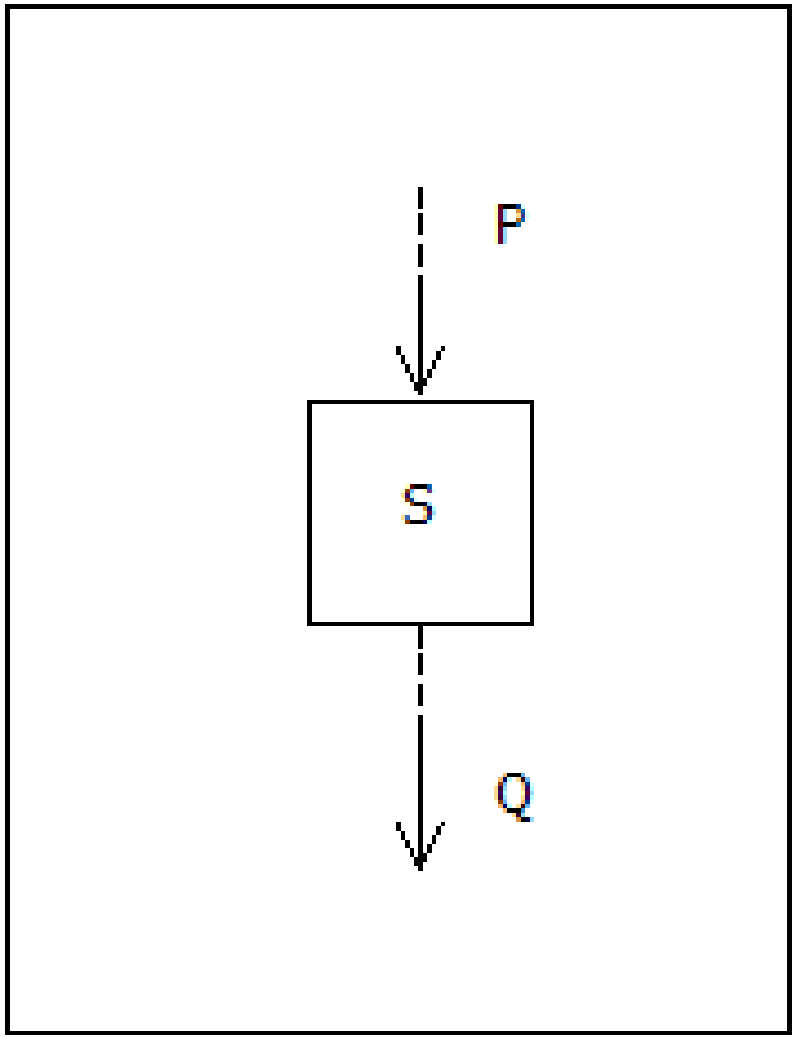
Object – Oriented Technology

Chapter 2: Assertions

Suad Alagić

Springer 2016

PARTIAL CORRECTNESS



SAMPLE SPECIFICATION

$(0 \leq i) \wedge (i \leq j) \wedge (j < a.Length);$
{ int s = 0; int n=i;
  while $(n \leq j)$
  s += a[n]; n++;
}
$s = sum\{int\ k \in (i..j)); a[k]\ \};$

ASSIGNMENT

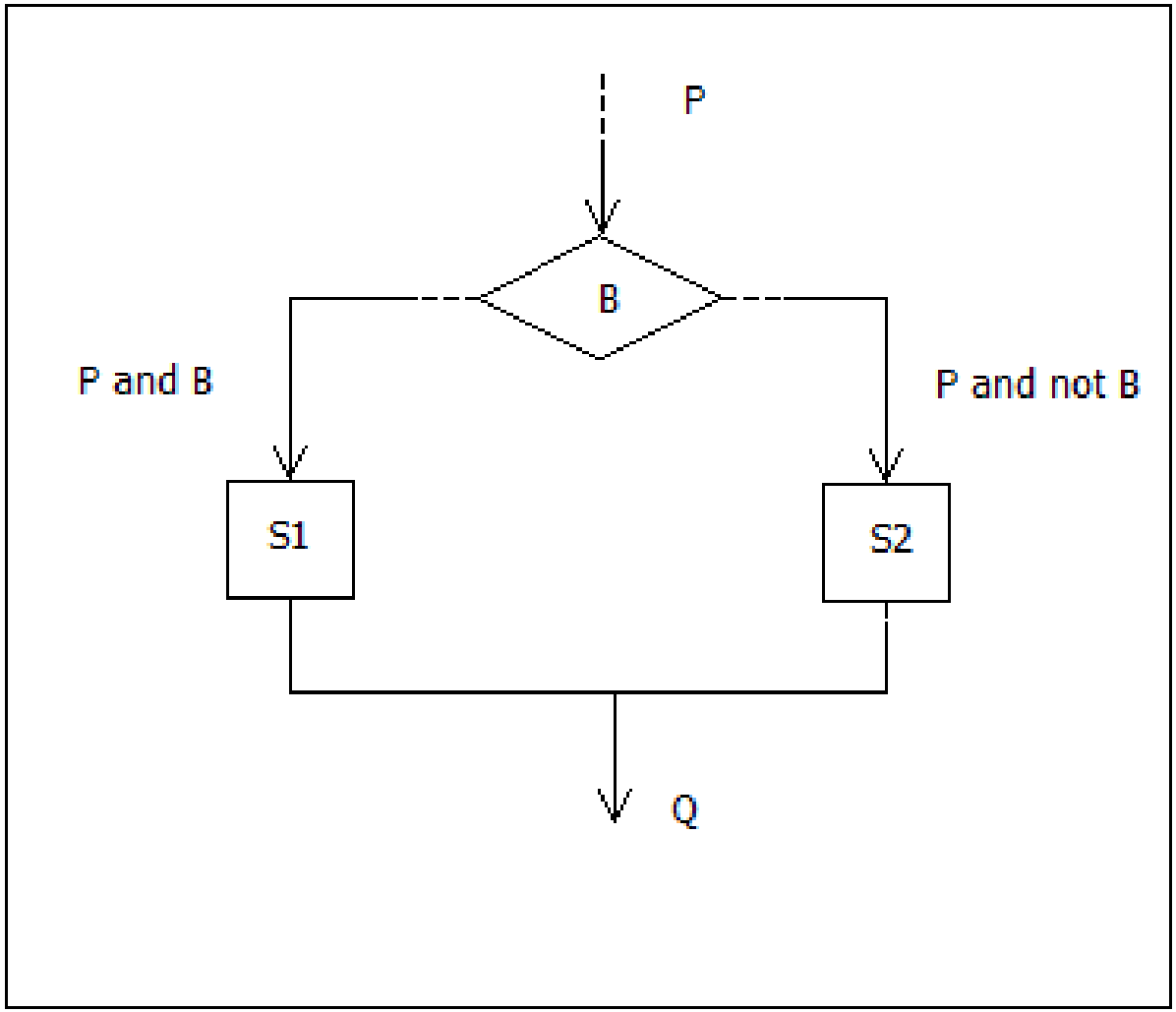
$$P[e/x]\{x=e\}P$$

$$x \geq 0 \ \{x=x+1\} \ x > 0$$

P is $x > 0$,

P[x+1/x] is $x+1 > 0$, i.e. $x \geq 0$.

CONDITIONAL STATEMENT



CONDITIONAL RULE

$$\frac{P \wedge B \{S1\} Q,\ P \wedge \neg B \{S2\} Q}{P \{\textbf{if } (B)\ S1\ \textbf{else } S2\ \} Q}$$

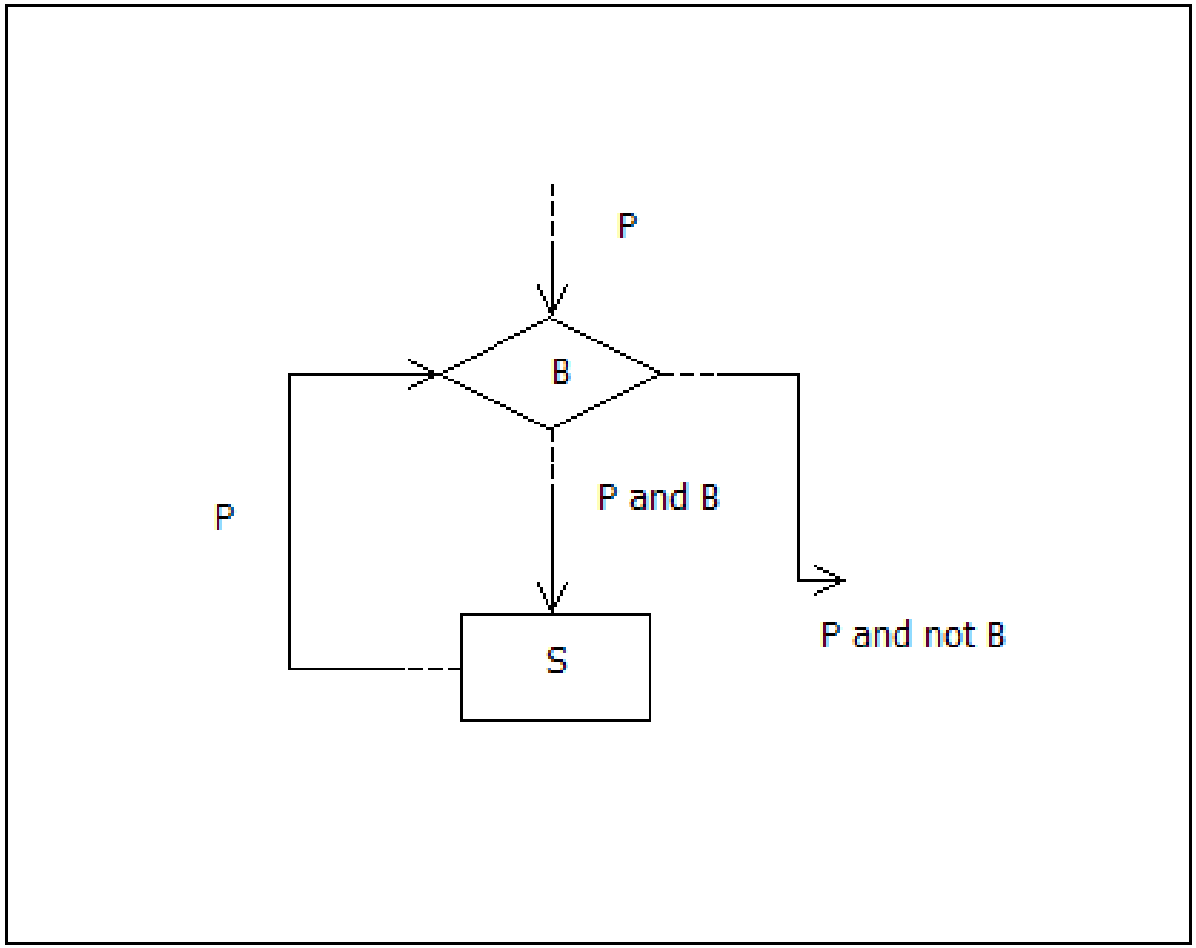
CONDITINAL EXAMPLE

$(x \neq 0)$**{if** $(x > 0)$ y=x **else** y= -x } $(y > 0)$

P: $(x \neq 0)$
Q: $(y > 0)$
P $\wedge$ B: $(x \neq 0) \wedge (x > 0)$
P $\wedge \neg$ B: $(x \neq 0) \wedge (x \leq 0)$
$(x \neq 0) \wedge (x > 0) \Rightarrow (x > 0)$
$(x \neq 0) \wedge (x \leq 0) \Rightarrow (x < 0)$

$(x > 0)$ {y=x} $(y > 0)$
$(x < 0)$ {y= -x} $(y > 0)$

WHILE LOOP



WHILE RULE

$$\frac{P \wedge B \ \{S\} \ P}{P \ \{\textbf{while} \ (B) \ S \ \} \ P \wedge \neg B}$$

WHILE EXAMPLE

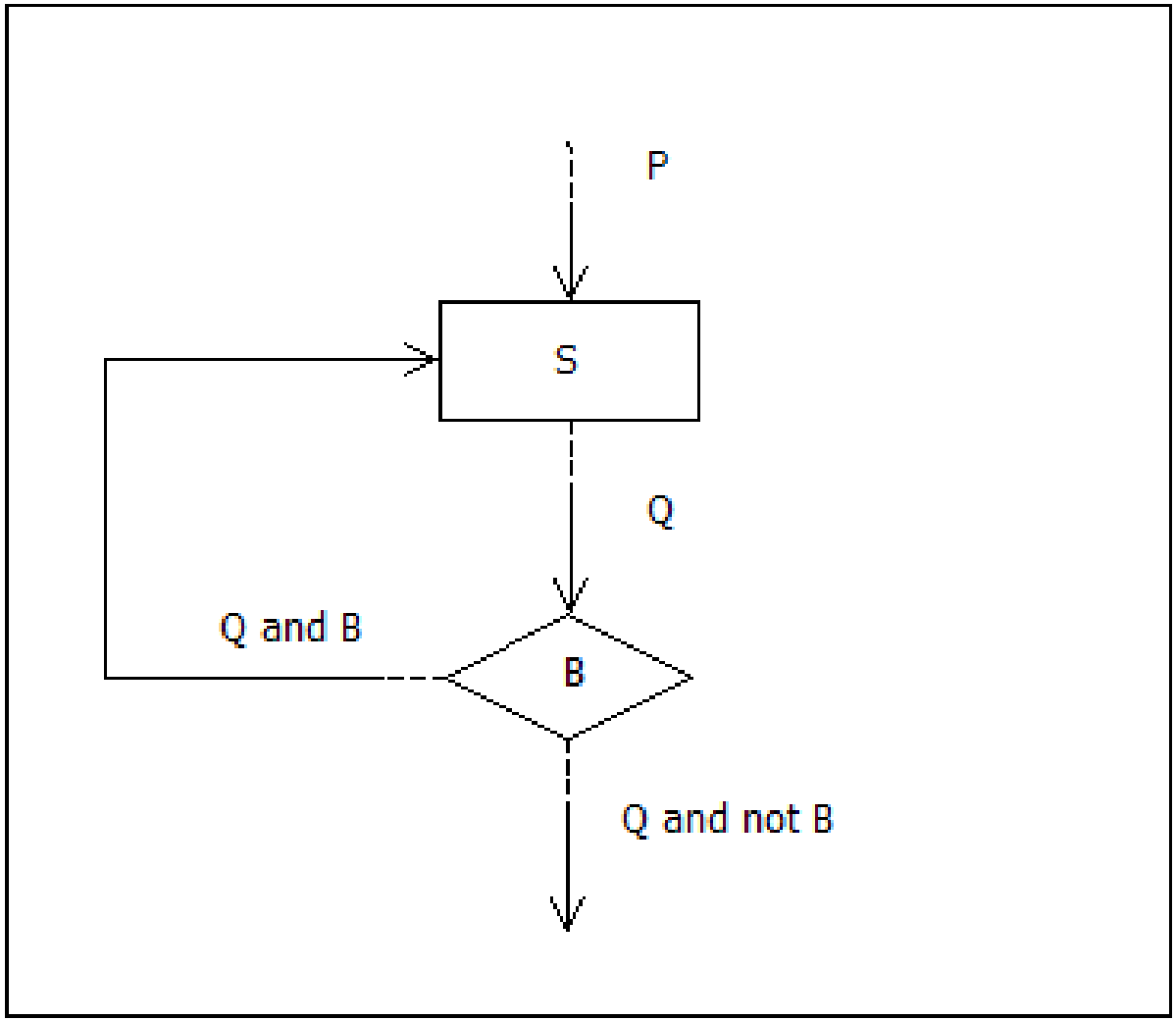
// precondition: $x \geq 0$;
{r=0;
while $((r+1)*(r+1) \leq x)$
// invariant: $r*r \leq x$;
  { r=r+1; }
}
// postcondition: $r*r \leq x \wedge x < (r+1)*(r+1)$

WHILE EXAMPLE

$(r * r \leq x) \wedge (x \geq (r+1) * (r+1))$
{ r=r+1; }
$(r * r) \leq x$

$(r * r) \leq x < (r+1) * (r+1)$

DO LOOP



DO RULE

$$\frac{P\ \{S\}\ Q,\ Q \wedge B \Rightarrow P}{P\ \{\textbf{do}\ S\ \textbf{while}\ (B)\}\ Q \wedge \neg B}$$

DO EXAMPLE

```
// precondition: a.Length ≥ 0
{int n = a.Length;
do
// invariant: (0 ≤ n) ∧ (n ≤ a.Length);
// invariant: ∀ { int i (n ≤ i) ∧ (i < a.Length); a[i] ≠ key};
  { n=n-1;
        if (n < 0) {
        break;
        }
  } while (a[n] ≠ key);
}
// postcondition (n ≥ 0) ⇒ (a[n] = key)
```

```
public interface Collection<T>
{ public boolean contains(T obj);
  public void add(T obj)
  ensures this.contains(obj);
  public void remove(T obj)
  requires this.contains(obj);
// ...
}
```

```
public interface Set<T>
                                  extends Collection<T>
{ public Set<T> union(Set<T> S);
  public Set<T> intersection(Set<T> S);
  // ...
invariant
(∀ Set<T> this, S; ∀ T: obj)
(this.union(S).contains(obj) ⇐ this.contains(obj);
this.union(S).contains(obj) ⇐ S.contains(obj);
this.intersection(S).contains(obj) ⇐
                this.contains(obj) ∧ S.contains(obj));
// ...
}
```

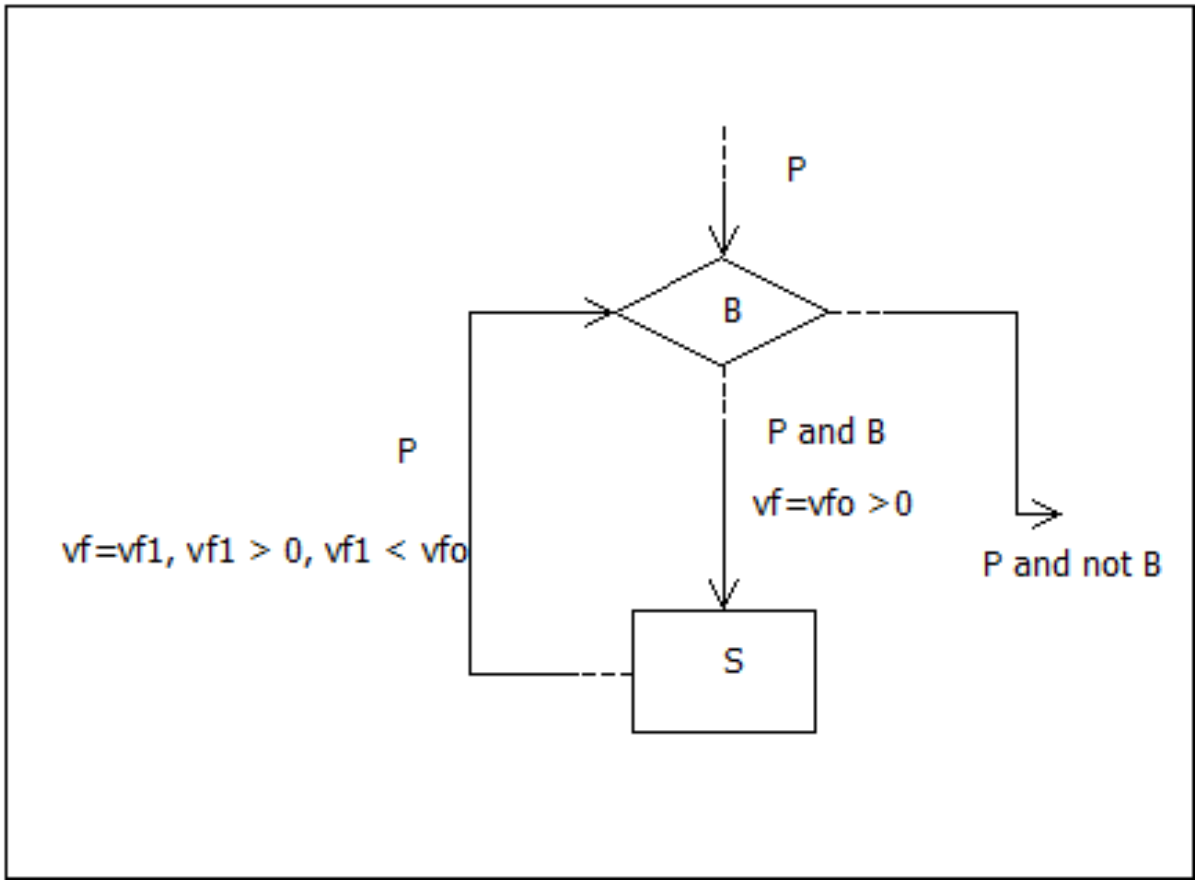
PRECONDITIONS and POSTCONDITIONS

void exchangeElements(int[] a, int i, int j)
requires $(0 \leq i) \wedge (i < a.Length)$;
requires $(0 \leq j) \wedge (j < a.Length)$;
modifies a[i], a[j];
ensures a[i] = **old**(a[j]);
ensures a[j] = **old**(a[i]);
{ int temp;
  temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}

LOOP INVARIANTS

```
public int ArraySum(int[]! a)
ensures result = sum {int i ∈ (0: a.Length); a[i]};
{ int s = 0;
  for (int n = 0; n < a.Length; n++)
        invariant n ≤ a.Length;
        invariant s = sum {int i ∈ (0: n); a[i] };
{ s += a[n];}
  return s;
}
```

TERMINATION



```
public int ArrayRangeSum(int[]! a, int i, int j)
requires (0 ≤ i) ∧ (i ≤ j) ∧ (j ≤ a.Length);
ensures result = sum {int k ∈ (i: j); a[k] };
{ int s = 0; int n=i;
  while (n < j)
  invariant (i ≤ n) ∧ (n ≤ j);
  invariant s = sum {int k ∈ (i: n); a[k]};
  invariant 0 ≤ j - n;
  { int variant = j - n;
  s += a[n]; n++;
  assert j - n < variant;
  }
return s;
}
```

OBJECT INVARIANTS

```
class Accumulator {
  int count;
  invariant count ≥ 0;
  public Accumulator()
  { count = 0; }
  public void Inc(int increase)
  modifies count;
  ensures count = old( count)+increase;
  { expose (this) {
        count = count + increase;
        }
  }
}
```

OBJECT INVARIANTS

```
public void Inc(int increase)
modifies count;
requires increase > 0;
ensures count = old( count)+increase;
{ expose (this) {
        count = count + increase;
        }
}
```

ASSERTIONS FOR COLLECTIONS

```
public abstract class CollectionSpec<T>{

  [SpecPublic] protected List<T!>!
        elements = new List<T!>();
  [SpecPublic] protected int size = 0;

  invariant elements.Count = size;
  invariant size ≥ 0;

  [Pure] public bool contains(T! x)
  ensures result = elements.Contains(x);
  { code }

  public virtual void add(T! x)
  modifies elements;
  ensures this.contains(x);
  { code }
```

CollectionSpec cont.

```
  public virtual void remove(T! x)
  modifies elements;
  requires this.contains(x);
  { code }
}
```

ASSERTIONS for BAGS

```
public class BagSpec<T> : CollectionSpec<T>
{
  [Pure] public int occurrences(T! x)
  ensures result = Count{int i ∈ (0: elements.Count);
        x.equals(elements[i])};
  { code }

  public override void add(T! x)
  modifies elements;
  ensures occurrences(x) = old(occurrences(x)) + 1;
  { code }

  public override void remove(T! x)
  modifies elements;
  ensures occurrences(x) = old(occurrences(x)) - 1;
  { code }
```

ASSERTIONS for BAGS

[Pure] public BagSpec<T!>! union(BagSpec<T!>! other)
ensures result.contains(x) $\Leftrightarrow$
this. contains(x) $\vee$ other.contains(x);
// type of x inferred as T
ensures this.contains(x) $\wedge$ other.contains(x) $\Rightarrow$
result.occurrences(x)=
Max{this.occurrences(x), other.occurrences(x)};
{code }

ASSERTIONS for BAGS

[Pure] public BagSpec<T!>! intersection(BagSpec<T!>! other)
ensures result.contains(x) $\Leftrightarrow$
this.contains(x) $\wedge$ other.contains(x);
ensures result.occurrences(x)=
Min{this.occurrences(x), other.occurrences(x)};
{code }

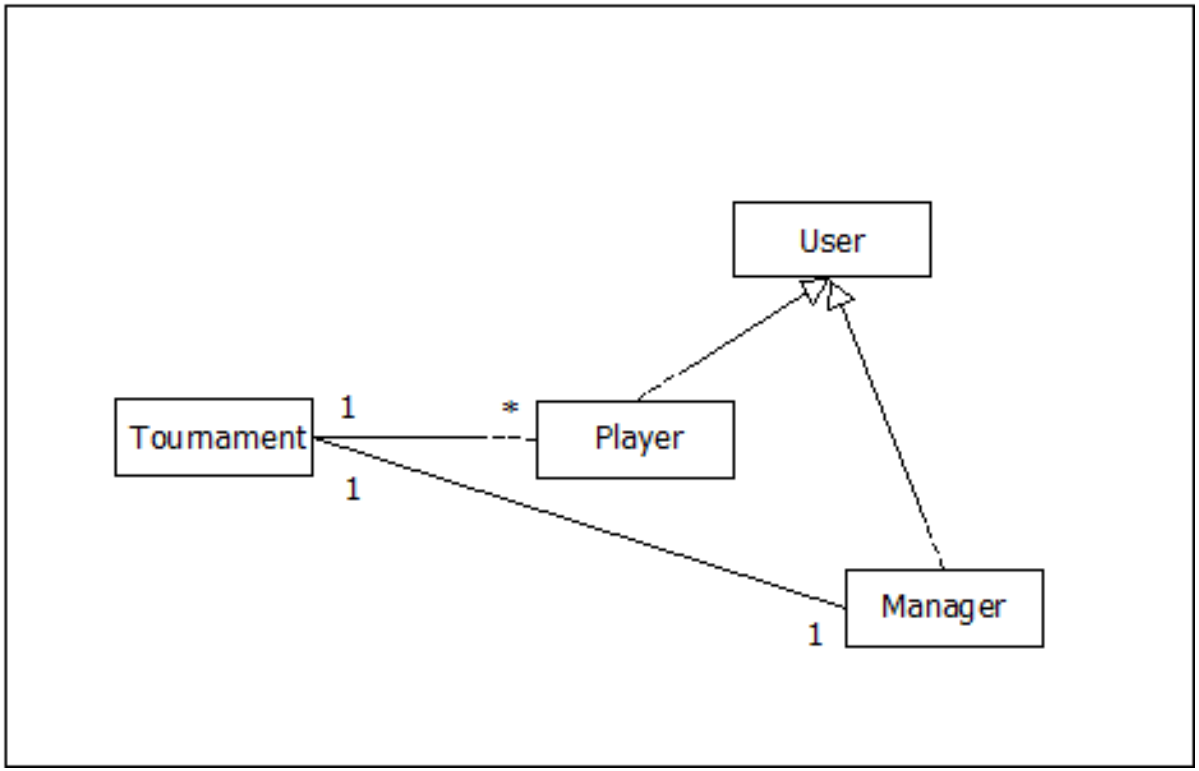
```
public class Stock {
  private String code;
  public String getCode()
  { return code; }
  invariant this.getCode() ≠ null;
  [SpecPublic] protected int value;
  public int getValue(){ return value; }
  public virtual void setValue(int v)
  ensures value=v;
  { value=v; }
}
```

```
public class BackupStock: Stock {
  [SpecPublic] protected int backup;
  public override void setValue (int v)
  ensures backup =old(value);
  { backup=value; value=v; }
}
```

BEHAVIORAL SUBTYPING

```
public class GrowingStock: Stock {
  public override void setValue(int v)
  requires value ≤ v; //not allowed
  { base.setValue(v);}
}
```

TOURNAMENT APPLICATION



TOURNAMENT APPLICATION

```
class Tournament {
  String name;
  Manager manager;
  List<Player> players = new List<Player>();
  // other fields
  // constructor
  public String Name
  {       get { return name; }
          set { name = value; }
  }
  // other poperties
}
```

```
abstract class User {
  String IDNum;
  String name;
  String role;
  [ContractInvariantMethod]
  void ObjectInvariant() {
        Contract.Invariant(this.UserName ≠ null);
        Contract.Invariant(this.ID ≠ null); }
public String ID
  {     get { return IDNum; }
        set { IDNum = value; } }
  public String UserName
  {     get { return name; }
        set { user = value; }}
  // Role property
}
```

```
class Player : User {
  int winCount;
  public int WinCount
  {       get { return winCount; }
          set { winCount = value; }
  }
  //other properties
  [ContractInvariantMethod]
  void ObjectInvariant() {
          Contract.Invariant(this.WinCount ≥ 0);
  }
  // constructor and other methods
}
```

```
class Manager : User {
  // fields
  [ContractInvariantMethod]
  void ObjectInvariant() {
Contract.Invariant(Role.ToUpper().Contains("MANAGER"));
  }
// methods
}
```

```
[Pure]
public boolean playerRegistered(Player newPlayer,
                    Tournament tournament) {
  foreach (Player player in tournament.players)
  { if (newPlayer.UserName.ToUpper().Equals(
                    player.UserName.ToUpper()))
          return true;
  }
  return false;
}
```

TOURNAMENT APPLICATION

```
Contract.Requires(newPlayer ≠ null);
Contract.Requires(tournament ≠ null);

Contract.Requires(¬ playerRegistered(newPlayer,
tournament));

Contract.Ensures((tournament.Players.Count) =
    (Contract.OldValue(tournament.Players.Count) + 1));
```

TOURNAMENT APPLICATION

```
public void addPlayer(Player newPlayer, Tournament tournament) {
Contract.Requires(newPlayer ≠ null);
Contract.Requires(tournament ≠ null);
Contract.Requires(¬ playerRegistered(newPlayer, tournament));
Contract.Ensures(playerRegistered(newPlayer, tournament));
Contract.Ensures((tournament.Players.Count) =
  (Contract.OldValue(tournament.Players.Count) + 1));
tournament.Players.Add(newPlayer);
}
```

Contract.Invariant(Contract.ForAll(Players, p $\Rightarrow$ p $\neq$ **null**));

Contract.Invariant(Contract.Exists(Players, p $\Rightarrow$
p.WinCount $>$ 0));

TOURNAMENT APPLICATION