# Software Code Review

Software Engineering
Prof. Maged Elaasar

# Learning objectives

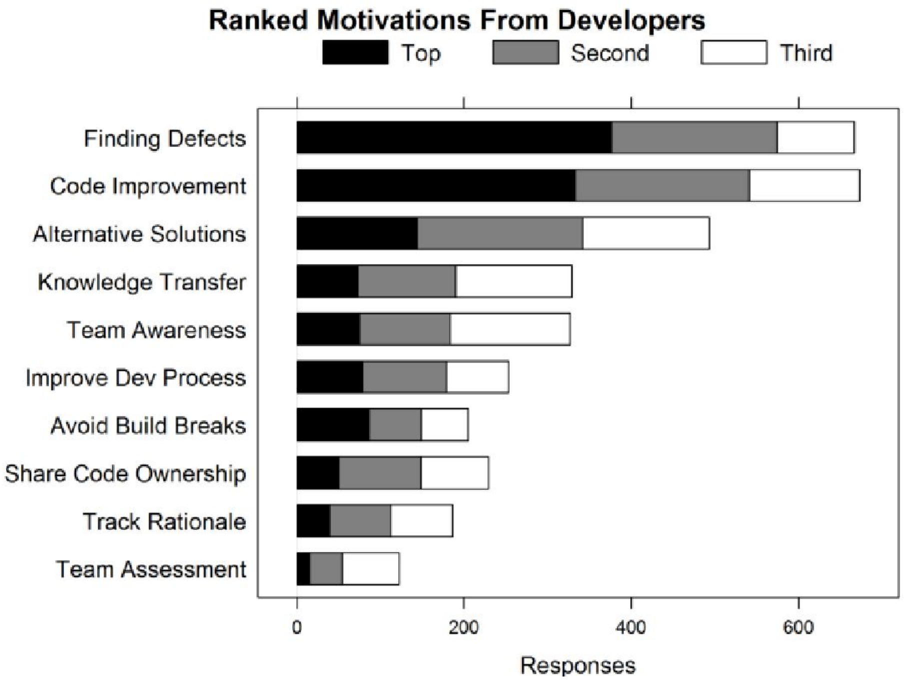
- Code reviews practices
- Hoare Logic: a formal code review technique

# Code Review

# When should you approve a code change?

- If it does not break any existing (regression) tests.
- If it conforms to the style guide (formatting/indentation).
- If it does not break the existing program modularity.
- If it documents the APIs (input / output).
- If it documents the design decisions.
- If it preserves/improves non-functional aspects.
- If it has accompanied tests with good coverage
- If it asserts weakest preconditions and/or strongest postconditions

# What is the motivation for code reviews?




EXPECTATIONS, OUTCOMES, AND CHALLENGES OF MODERN CODE REVIEW, ICSE 2013, BACCHELLI AND BIRD

# Level of code understanding for effective review

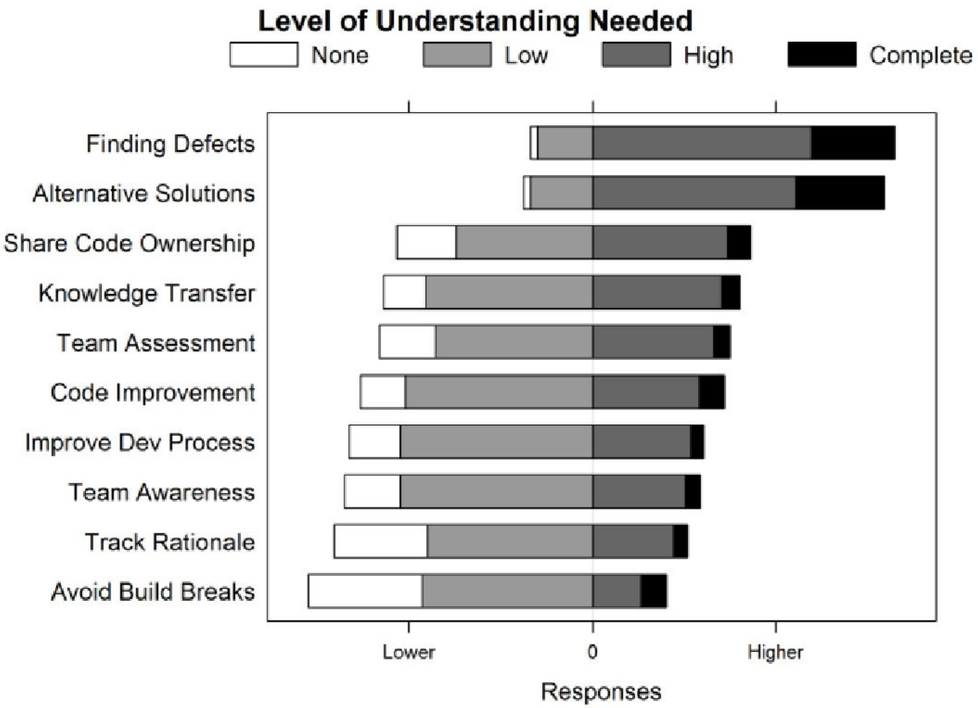



*Figure 5. Developers' responses in surveys of the amount of code understanding for code review outcomes.*

# Modern code review practices

- Most reviews do not actually find bugs
    - 15% of comments indicate a possible bugs.
    - 50% of comments are on syntax and style conformance issues
    - 33% of comments are deemed useful by the author.

- New reviewers learn fast but need at least 6-12 months to be productive as the rest of the team

- We need to have rigorous criteria for code reviews to make them effective.

- Developers need to have critical eyes for reviewing code changes, thinking about corner cases.

# Take away message

- Code reviews are time consuming but developers provide shallow style feedback and are not effective at finding bugs.
- Developers need to have “critical eyes” for reviewing code changes, thinking about corner cases.
- **Hoare Logic** is a formal method for code review that can help students become effective code reviewers.

# Hoare Logic

# Hoare Logic: a formal code review technique

- Logical rules for reasoning about the correctness of computer programs
- Helps code comprehension, collaboration, review, and bug finding.
- Involves writing **preconditions** and **postconditions** on functions and reasoning about code correctness using them
- In practice, programmers do not write pre-/post conditions, or write very shallow ones.
- Therefore, we are teaching **Hoare Logic** so that you can take advantage of this formal method to improve code or do a more effective code review

# What is Hoare Logic good for?

- Suppose that your colleague wrote the code for a function
- Which input arguments do you need to have, to not crash her code?
- Computing weakest preconditions can find **subtle bugs** and **corner cases** during **code reviews.**

# Code review scenario

```
public char[] foo(Object x, int z) {
    if (x != null) {
      n = x.f;
    } else {
      n = z-1;

      z++;
    }
    a = new char[n];
    return a;
}
```

Suppose Alice wrote `foo`. **Which arguments need to be passed to** `foo` so that it returns a non-null value without throwing any exception?

# Predicate

A predicate is a Boolean function on the program state

Examples:

- $x == 8$
- $x < y$
- $m \leqq n \Rightarrow (\forall j \mid 0 \leqq j < a.length \cdot a[j] \neq NaN)$
- true
- false

# Hoare triples

- For any predicate **P** and **Q** and any program **S**, the $\boxed{\{P\}\ S\ \{Q\}}$ Hoare triple says that if **S** is started in (a state satisfying) **P**, then it terminates in (a state satisfying) **Q**.

- Examples
  - {true} x := 12 {x=12}
  - {x < 40} x :=12 {x >= 10}
  - {x < 40} x:=x+1 {x <=40}
  - {m <= n} j := (m+n)/2 {m <= j <= n}

- If {**P**} **S** {**Q**} and {**P**} **S** {**R**}, then does {**P**} **S** {**Q and R**} hold?? … yes!

- The most precise **Q** such that {**P**} **S** {**Q**} is called the strongest postcondition of **S** with respect to **P**.

# Weakest precondition

- If {**P**} **S** {**R**} and {**Q**} **S** {**R**}, then {**P** or **Q**} **S** {**R**} holds.
- The most general **P** such that {**P**} **S** {**R**} is called the weakest precondition of S with respect to **R**, written wp(**S**, **R**)
- In fact, {**P**} **S** {**Q**} holds if and only if **P** ⇒ wp(**S**, **Q**)

Note: ⇒ is the logical “**implies**” operator

# Program semantics: skip

- no-op
- $wp(\text{skip}, R) \equiv R$
- $wp(\text{skip}, x^n + y^n = z^n) \equiv x^n + y^n = z^n$

# Program semantics: assert

- assert P means if P holds, do nothing, else terminate
- wp(assert P, R) ≡ P **and** R

- wp(assert x<10, x>=0) ≡ x<10 and x>=0 ≡ 0<=x<10
- wp(assert x=y*y, x>=0) ≡ x=y^2 and x>=0 ≡ x=y^2
- wp(assert false, x<=10) ≡ false and x<=10 ≡ false
  (precondition being **false** means that there exists no input value that will execute the program statement and ends up with a postcondition x<=10)

# Program semantics: assignment

- Evaluate E and change value of w to E
- $wp(w := E, R) \equiv R[w := E]$

replace w
by E in R

- $wp(x := x+1, x<=10) \equiv \{x+1<=10\} \equiv \{x<=9\}$
- $wp(x := 15, x <= 10) \equiv \{15 <=10\} \equiv false$
  (there is no input that will go through $x:=15$ and end up with postcondition $x<=10$)

- $wp(y := x + 3*y, x<=10) \equiv \{x<=10\}$
- $wp(x,y := y,x,\ x<y) \equiv \{x>y\}$

Shorthand for
swapping values

# Program semantics: sequential composition

- $wp(S;T, R) \equiv wp(S, wp(T, R))$

- $wp(x := x+1 \; ; \; \text{assert } x <= y, \; x<0) \equiv wp\ (x:=x+1, \; wp\ (\text{assert } x<=y, \{x<0\}))$
  $\equiv wp\ (x:=x+1, \{x<=y \text{ and } x<0\}) \equiv \{x+1<=y \text{ and } x+1<0\} \equiv \{x+1<=y \text{ and } x<-1\}$

- $wp(y := y+1 \; ; \; x := x + 3*y, \; y <= 10 \text{ and } 3 <= x)$
  $\equiv wp\ (y:=y+1, \; wp\ (x:=x+3y, \{y<=10 \text{ and } 3<=x\}))$
  $\equiv wp\ (y:=y+1, \{y<=10 \text{ and } 3<=x+3y\})$
  $\equiv \{y+1<=10 \text{ and } 3<=x+3(y+1)\}$
  $\equiv \{y<=9 \text{ and } 3<=x+3y+3\}$
  $\equiv \{y<=9 \text{ and } 0<=x+3y\}$

# Program semantics: if-else condition

- wp(if B then S else T end, R) ≡ (B and wp(S, R)) or (!B and wp(T, R))

- wp(if x < y then z:=y else z:=x end,, z>=0)
  ≡ {x<y and wp(z:=y, z>=0)} or {x>=y and wp(z:=x, z>=0)}
  ≡ {x<y and y>=0} or {x>=y and x>=0}

- wp(if x!=10 then x:=x+1 else x:=x+2 end, x<=10)
  ≡ {x!=10 and wp(x:=x+1, x<=10)} or {x==10 and wp(x:=x+2, x<=10)}
  ≡ {x!=10 and x+1<=10} or {x==10 and x+2<=10}
  ≡ {x<=9} or {x==10 and x<=8}
  ≡ {x<=9} or false ≡ {x<=9} // there is something wrong with the code, and it's not possible to go through else side and produce a desired post condition.

# Program semantics: if condition only

- wp(if B then S end, R) ≡ (B and wp(S, R)) or (!B and R)

- wp(if x < 1 then x:=++; end,, x=0)
  ≡ {x<1 and wp(x++, x=0)} or {x>=1 and x=0}
  ≡ {x<1 and x+1=0} or false
  ≡ {x<1 and x=-1}
  ≡ x=-1

# Review

- $wp(\text{skip}, R) \equiv R$
- $wp(\text{assert } P, R) \equiv P \text{ and } R$
- $wp(w := E, R) \equiv R[w := E]$
- $wp(S;T, R) \equiv wp(S, wp(T, R))$
- $wp(\text{if } B \text{ then } S \text{ else } T \text{ end}, R) \equiv (B \text{ and } wp(S, R)) \text{ or } (!B \text{ and } wp(T, R))$
- $wp(\text{if } B \text{ then } S \text{ end}, R) \equiv (B \text{ and } wp(S, R)) \text{ or } (!B \text{ and } R)$

# Exercise 1: use Hoare Logic to review this code

```
public char[] foo(Object x, int z) {
    if (x != null) {
      n = x.f;
   } else {
      n = z-1;
      z++;
    }
    a = new char[n];
    return a;
}
```

# Calculate the weakest precondition

★ which precondition should hold here?

```
if (x != null) {
    n = x.f;
 } else {
    n = z-1;
    z++;
 }
 a = new char[n];
```

★ If the postcondition is **true**

# Solution

```
(x!=null and x.f>=0) or (x==null and z-1>=0)
    if (x != null) {
x.f>=0
        n = x.f;
    } else {
z-1>=0
        n = z-1;
n>=0
        z++;
      }
n>=0
      a = new char[n];
true
```

IndexOutOfBoundsException
if n<0

# What do we do with weakest preconditions?

- Add assertions at a right point during debugging.
  (You can later turn them off if performance is an issue)

- Create test cases that satisfy weakest preconditions and expect to pass.

- Also create test cases that violate weakest preconditions and expect to fail.

# Exercise 2

```
if (x!= null) {
   n =x.f;
} else {
   n = z+1;
   z = 2*z+1;
 }
 a = new char[n-2];
 c = a[z];
```

Suppose Tom wrote this code. What is the weakest precondition such that this code terminates **without throwing any exceptions**?

# Solution

```
{x!=null and wp(n:=x.f, n>=2 and n>z+2 and z>=0} or {x==null and wp(n:=z+1;z:=2z+1, n>=2 and n>z+2 and z>=0)}
≡ {x!=null and x.f>=2 and x.f>z+2 and z>=0} or {x==null and wp(n:=z+1, n>=2 and n>2z+1+2 and 2z+1>=0)}
≡ {x!=null and x.f>=2 and x.f>z+2 and z>=0} or {x==null and z+1>=2 and z+1>2z+3 and 2z+1>=0)}
≡ {x!=null and x.f>=2 and x.f>z+2 and z>=0} or {x==null and z>=1 and -2>z and 2z+1>=0)
≡ {x!=null and x.f>=2 and x.f>z+2 and z>=0} or {x==null and FALSE and 2z+1>=0)}
≡ {x!=null and x.f>=2 and x.f>z+2 and z>=0}
  if (x!= null) {
     n =x.f;
   } else {
     n = z+1;
     z = 2*z+1;
   }
wp(a:=new char[n-2], z<a.length and z>=0)
≡ {n-2>=0 and z<n-2 and z>=0}
≡ {n>=2 and n>z+2 and z>=0}
   a = new char[n-2];
wp(c = a[z], true)
≡ z>=0 and z<a.length
   c = a[z];
true
```

# Code Review Quiz

# Reasoning about loops

{P} while B do S end {Q}

Find a loop invariant **J** and a variant function **vf** such that:

1. **J holds initially**: P ⇒ J
2. **J is maintained**: {J and B} S {J}
3. **J is sufficient**: (J and !B) ⇒ Q
4. **vf is bounded**: (J and B) ⇒ (vf>=0)
5. **vf decreases**: {J and B and vf=VF} S {vf<VF}

# Exercise 3: Array Sum

```
k := 0;
s := 0;
while k!=n do
    s:=s+a[k]
    k:=k+1
end
```

Suppose that Tom wrote the above code to compute the sum of integer elements in the array. How can he prove its correctness?

# Exercise 3: Array Sum

P: **n>=0**

```
k := 0;
s := 0;
while k!=n do
    s:=s+a[k]
    k:=k+1
end
```

Guess the loop invariant **J** and variant function **vf** such that:

1. J initially holds: P ⇒ J
2. J is maintained: {J and B} S {J}
3. J is sufficient: J and !B ⇒ Q
4. vf is bounded: (J and B) ⇒ (vf>=0)
5. vf decreases: {J and B and vf=VF} S {vf<VF}

Q: **s = (Σi | 0<=i<n · a[i])**

# Tips for loop reasoning

- How do you guess a loop invariant?
  - Generally by modifying the post condition and the loop guard.
- How do you guess a variant function?
  - Generally by coming up with a function n-k if k increases by 1.
- Can we automatically find loop invariants?
  - Yes, it’s an active research area to find a loop invariant. Mostly they generate candidates that need to be verified.
- Can we infer pre and post conditions?
  - Yes, There is also work that infers pre/post conditions from tests

# Exercise 3: Array Sum

P: **n>=0**

```
k := 0;
s := 0;
while k!=n do
    s:=s+a[k]
    k:=k+1
end
```

Guess Invariant J and a variant function vf as:
J: **s = (Σi | 0<=i<k · a[i]) and 0<=k<=n**
vf: **n-k**

Q: **s = (Σi | 0<=i<n · a[i])**

# Loop invariant caveats

- You will need to guess the loop invariant J first then show that your guess satisfies the three loop invariant conditions:
    - J initially holds: P ⇒ J
    - J is maintained: {J and B} S {J}
    - J is sufficient: J and !B ⇒ Q

- There could be more than one loop invariant.

- In the absence of a human-provided meaningful post-condition, the loop invariant subsequently is often not meaningful as well.

# Exercise 3: Array Sum

**Step 1:** Invariant holds initially (before the loop): **P ⇒ J**

$\{0 <= n\}$

$\Rightarrow$

$\equiv \{0 <= n\}$

$\{0 = (\Sigma i \mid 0 <= i < 0 \cdot a[i]) \text{ and } 0 <= 0 <= n\}$

  **k := 0;**

$\{0 = (\Sigma i \mid 0 <= i < k \cdot a[i]) \text{ and } 0 <= k <= n\}$

  **s := 0;**

$\{s = (\Sigma i \mid 0 <= i < k \cdot a[i]) \text{ and } 0 <= k <= n\}$

# Exercise 3: Array Sum

**Step 2:** Invariant is maintained (throughout the loop): **{J and B}** **S** **{J}**

≡ {s = (Σi | 0<=i<k · a[i]) and 0<=k<=n and k!=n}

≡ {s = (Σi | 0<=i<k · a[i]) and 0<=k<=n and k!=n}

{s+a[k] = (Σi | 0<=i<k · a[i])+a[k] and 0<=k<n}

**s := s + a[k];**

≡ {s = (Σi | 0<=i<k · a[i])+a[k] and -1<=k<n}

{s = (Σi | 0<=i<k+1 · a[i]) and 0<=k+1<=n}

**k := k+1;**

{s = (Σi | 0<=i<k · a[i]) and 0<=k<=n}

due to a[k]

s = (Σi | 0≦i<k+1 · a[i])

= a[0] + a[1] … a[k-1] +a[k]

= **sum 0<=i<k a[i]**+a[k]

# Exercise 3: Array Sum

**Step 3:** Invariant is sufficient: **{J and !B}** ⇒ **Q**

{s = (Σi | 0<=i<k · a[i]) and 0<=k<=n and !(k!=n)}
≡ {s = (Σi | 0<=i<k · a[i]) and 0<=k<=n and k==n}
≡ {s = (Σi | 0<=i<n · a[i]) and 0<=n}
⇒
{s = (Σi | 0<=i<n · a[i])}

Notice that the following is logically true:
X and Y ⇒ X

# Variant function caveats

- First, Just like a loop invariant, vf is a piece of puzzle that you need to complete an inductive proof. So its role is similar to an inductive case where you have to guess first and show that your choice of vf satisfies the two conditions:
  - vf is bounded: J and B ⇒ (vf >= 0)
  - vf decreases: {J and B and vf = VF} S {vf < VF}

- vf is bounded: **J and B** ⇒ (**vf>=0**). This means that while the loop is running (i.e., **B** is **true** and **J** is also **true**), the variant function that you picked has a positive value.

- **vf** is a monotonically decreasing function that starts with value **VF** at the beginning of the loop and converges towards 0 when the loop terminates.

# Exercise 3: Array Sum

**Step 4:** function variant is bounded: **{J and B}** ⇒ **{vf>=0}**

{s = (Σi | 0<=i<k · a[i]) and 0<=k<=n and (k!=n)}
≡ {s = (Σi | 0<=i<k · a[i]) and 0<=k<n}
≡ {s = (Σi | 0<=i<k · a[i]) and 0<=k and k<n}
⇒
≡ {k<=n}
{n-k>=0}

Notice that the following is logically true:
k<n ⇒ k<=n
since k<=n is k<n or k==n (and the latter is false)

# Exercise 3: Array Sum

**Step 5:** function variant decreases: **{J and B and vf=VF} S {vf<VF}**

{s = (Σi | 0≦i<k · a[i]) and 0<=k<=n and k!=n and n-k=VF}
≡ {s = (Σi | 0≦i<k · a[i]) and 0<=k<n and n-k=VF}
≡ {s = (Σi | 0≦i<k · a[i]) and 0<=k<n and n-k-1<VF}
⇒
{n-k-1<VF}
  **s := s + a[k];**
≡ {n-k-1<VF}
{n-(k+1)<VF}
  **k := k+1;**
{n-k<VF}

Notice that the following is logically true:
n-k=VF ⇒ n-k-1<VF
since VF - 1 < VF

# Practice at home

```
public static int power(int x, int n) {
    int p = 1, i = 0;
    while (i < n) {
        p = p * x;
        i = i + 1;
    }
    return p;
}
```

- What is a loop invariant that must hold given a precondition n>=0?
- Give a proof to Sheryl's manager that after the execution of this loop, the return value p = x^n.

# Practice at home - solution

P: n>=0
Q: p=x^n
B: i<n
J: p=x^i and i<=n
vf: n-i

**Step 1: P ⇒ J**

{0<=n}
⇒
≡ {0<=n}
{1=1 and 0<=n}
    **p := 1;**
{p = x^0 and 0<=n}
    **i := 0;**
{p = x^i and i<=n}

**Step 2: {J and B} S {J}**

≡ {p=x^i and i<=n and i<n}
{p*x=x^i*x and i<n}
    **p := p * x;**
≡ {p=x^i*x and i<n}
{p=x^(i+1) and i+1<=n}
    **i := i+1;**
{p=x^i and i<=n}

**Step 3: {J and !B} ⇒ Q**

{p=x^i and i<=n and i>=n}
≡ {p=x^i and i==n}
≡ {p=x^n}
⇒
{p=x^n}

**Step 4:{J and B} ⇒ {vf>=0}**

{p=x^i and i<=n and i<n}
≡ {p=x^i and i<n}
⇒
≡ {i<=n}
{n-i>=0}

**Step 5: {J and B and vf=VF} S {vf<VF}**

{p=x^i and i<=n and i<n and n-i=VF}
≡ {p=x^i and i<n and n-i-1<VF}
⇒
{n-i-1<VF}
    **p := p * x;**
≡ {n-i-1<VF}
{n-(i+1)<VF}
    **i := i+1;**
{n-i<VF}

# Reflection

- It’s a **good practice** to think about what holds during a loop execution, and check whether that invariant is satisfied in the beginning and the end.

- If you guess a loop invariant. It’s good habit to write **assertions** during development.

- Why do we care to learn “loop invariants” and what are the implications?
  - Software verification tools automate the proof if you write pre/post conditions with invariants.
  - Writing contract is a very good practice!

# References


- Bacchelli, A., Bird, C.: “Expectations, Outcomes, and Challenges oF Modern Code Review.” ICSE 2013.
  https://sback.it/publications/icse2013.pdf

- Pierce, B., et. al: “Hoare Logic”. Programming Language Foundations. Software Foundations. volume 2, chapter 2, 2019.
  https://softwarefoundations.cis.upenn.edu/plf-current/Hoare.html