Debugging (1/2)

Martin Kellogg

Debugging (Part 1/2)

Today's agenda:

- **Finish static analysis slides**
- Reading Quiz
- What is a bug, anyway?
- Bug reports, triage, and the defect lifecycle
- Debugging
  - printf debugging and logging
  - delta debugging
  - debuggers

Debugging (Part 1/2)

Today's agenda:

- **Finish static analysis slides**
- Reading Quiz
- What is a bug, anyway?
- Bug reports, triage, and the def
- Debugging
  - printf debugging and logging
  - delta debugging
  - debuggers

Announcements:

- there is a midterm in this class one week from today
- if you want me to hold a review session, fill out the form I posted yesterday on Discord

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses
  - built into modern IDEs
  - aim for low false positive rates
  - widely used in industry:
    - ErrorProne at Google, Infer at Meta, SpotBugs at many places (including Amazon), Coverity, Fortify, etc.

Static analysis in practice

Less common, but useful to know about:

Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems

Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java

Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification*

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification*
  - you write a specification

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification*
  - you write a specification
  - tool verifies that code matches that specification

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification*
  - you write a specification
  - tool verifies that code matches that specification
  - very high effort, but enables sound reasoning about complex properties (= worth it for very high value systems)

Static analysis in practice: soundiness

- all “**sound**” static analyses have a ***trusted computing base*** (*TCB*)

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (***TCB***)
    - the TCB is the code whose correctness must be assumed for the analysis to actually be sound

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)

Static analysis in practice: soundness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)
  - TCB for some formal verifiers is **very small** (a few kLoC)
    - but these tools (e.g., Coq) are **much harder to use**

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)
  - TCB for some formal verifiers is **very small** (a few kLoC)
    - but these tools (e.g., Coq) are **much harder to use**
- soundness theorems also usually make some **assumptions** about the code being analyzed (e.g., no calls to native code, no reflection)

Static analysis: summary

- static analysis is very good at enforcing **simple rules**
  - **much** better than humans at this
- all interesting semantic properties of programs are **undecidable**, so all static analyses must **approximate**
  - goal in analysis design is to **abstract away unimportant details**, but keep important details
  - **dataflow analysis** is one technique for static analysis
  - trade-offs between false positives, false negatives, analysis time
- soundness & completeness are **possible, but rare**
  - all soundness guarantees come with caveats about the TCB

Debugging (Part 1/2)

Today's agenda:

- Finish static analysis slides
- **Reading Quiz**
- What is a bug, anyway?
- Bug reports, triage, and the defect lifecycle
- Debugging
    - printf debugging and logging
    - delta debugging
    - debuggers

Reading quiz: debugging (1)

Q1: What was wrong with the student email in the first reading?

A. the student assumed their guesses were correct
B. the student misused the debugger
C. the student didn't explain what they expected to happen
D. the email was too vague

Q2: **TRUE** or **FALSE**: the author of the second article argues that a debugger should be the first tool you reach for when debugging only in certain specific circumstances.

Reading quiz: debugging (1)

Q1: What was wrong with the student email in the first reading?

A. the student assumed their guesses were correct
B. the student misused the debugger
C. the student didn't explain what they expected to happen
D. the email was too vague

Q2: **TRUE** or **FALSE**: the author of the second article argues that a debugger should be the first tool you reach for when debugging only in certain specific circumstances.

Reading quiz: debugging (1)

Q1: What was wrong with the student email in the first reading?

A. the student assumed their guesses were correct
B. the student misused the debugger
C. the student didn't explain what they expected to happen
D. the email was too vague

Q2: **TRUE** or **FALSE**: the author of the second article argues that a debugger should be the first tool you reach for when debugging only in certain specific circumstances.

Debugging (Part 1/2)

Today's agenda:

- **What is a bug, anyway?**
- Bug reports, triage, and the defect lifecycle
- Debugging
  - printf debugging and logging
  - delta debugging
  - debuggers

Review: finding bugs

- Quality assurance is critical to software engineering

Review: finding bugs

- Quality assurance is critical to software engineering
- We've discussed **static** (code review, dataflow analysis) and **dynamic** (testing) approaches to finding bugs

Review: finding bugs

- Quality assurance is critical to software engineering
- We've discussed **static** (code review, dataflow analysis) and **dynamic** (testing) approaches to finding bugs
- Key question for today: what happens to all of the **bugs** those find?

Terminology: what is a bug?

Terminology: what is a bug?

- "**bug**" is an ambiguous term in common usage - it can refer to either static or dynamic problems

Terminology: what is a bug?

- "**bug**" is an ambiguous term in common usage - it can refer to either static or dynamic problems
- we'll use the following "standard" terms to disambiguate:

Terminology: what is a bug?

- “**bug**” is an ambiguous term in common usage - it can refer to either static or dynamic problems
- we’ll use the following “standard” terms to disambiguate:

Definition: a ***fault*** is an exceptional situation at run time

Terminology: what is a bug?

- “**bug**” is an ambiguous term in common usage - it can refer to either static or dynamic problems
- we’ll use the following “standard” terms to disambiguate:

Definition: a ***fault*** is an exceptional situation at run time

- when you’re running a program and something goes wrong, a fault has occurred

Terminology: what is a bug?

- "**bug**" is an ambiguous term in common usage - it can refer to either static or dynamic problems
- we'll use the following "standard" terms to disambiguate:

Definition: a ***fault*** is an exceptional situation at run time

- when you're running a program and something goes wrong, a fault has occurred

Definition: a ***defect*** is any characteristic of a product which hinders its usability for its intended purpose

Terminology: what is a bug?

- "**bug**" is an ambiguous term in common usage - it can refer to either static or dynamic problems
- we'll use the following "standard" terms to disambiguate:

Definition: a ***fault*** is an exceptional situation at run time

- when you're running a program and something goes wrong, a fault has occurred

Definition: a ***defect*** is any characteristic of a product which hinders its usability for its intended purpose

- cf. "design defect". I'll use "***bug***" to mean "a defect in source code"

Terminology: bug reports

Terminology: bug reports

Definition: a ***bug report*** provides information about a defect

Terminology: bug reports

Definition: a ***bug report*** provides information about a defect

- Created by testers, users, tools, etc.
- Often contains multiple types of information
- Often tracked in a database

Terminology: bug reports

Definition: a ***bug report*** provides information about a defect

- Created by testers, users, tools, etc.
- Often contains multiple types of information
- Often tracked in a database

Definition: A ***feature request*** is a potential change to the intended purpose (requirements) of software

Terminology: bug reports

Definition: a ***bug report*** provides information about a defect

- Created by testers, users, tools, etc.
- Often contains multiple types of information
- Often tracked in a database

Definition: A *feature request* is a potential change to the intended purpose (requirements) of software

- In CS: an ***issue*** is either a bug report or a feature request (cf. “issue tracking system”)

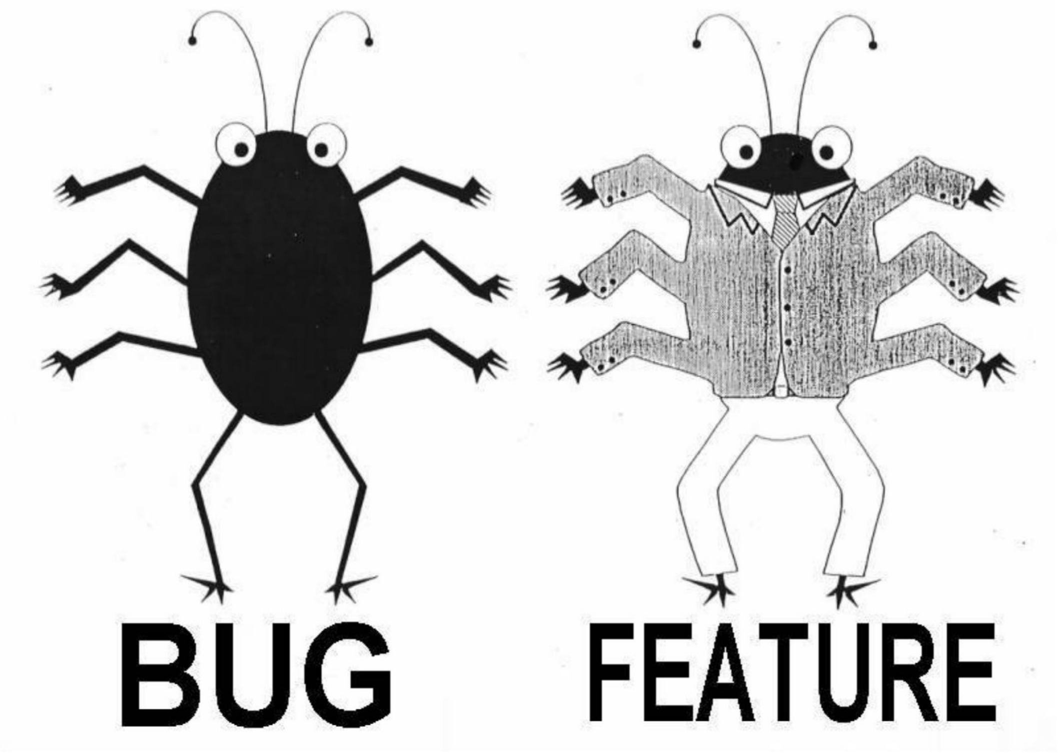
Terminology: bug vs. features

Terminology: bug vs. features

- what is a bug and what is a feature is **subjective**

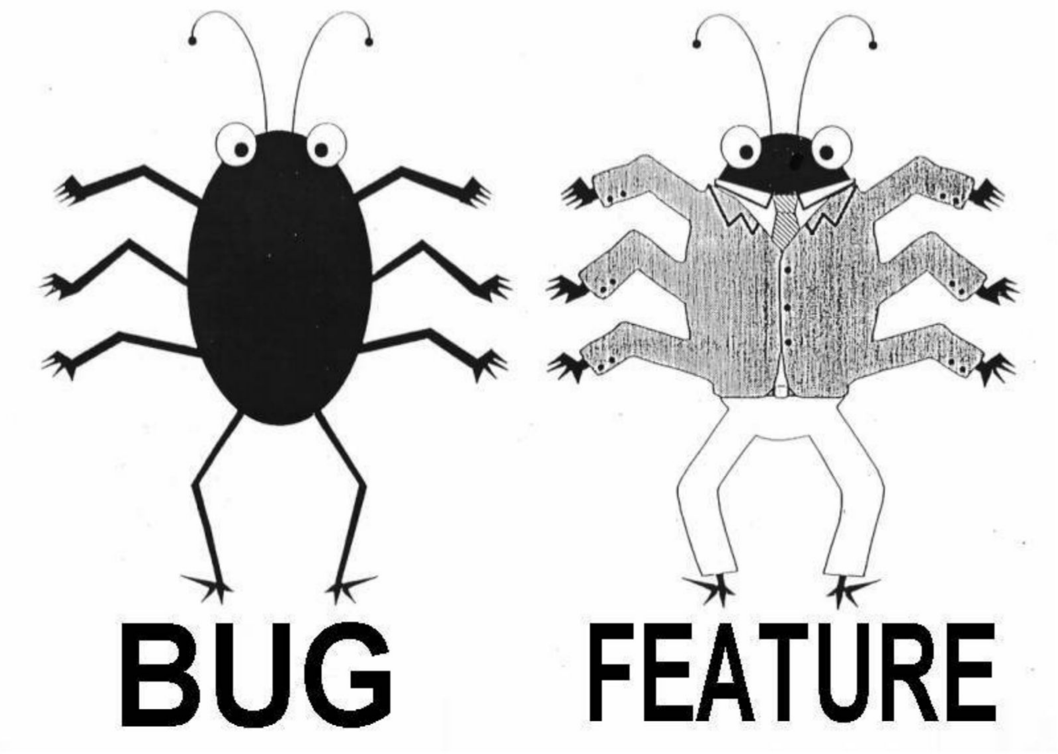
Terminology: bug vs. features

- what is a bug and what is a feature is **subjective**



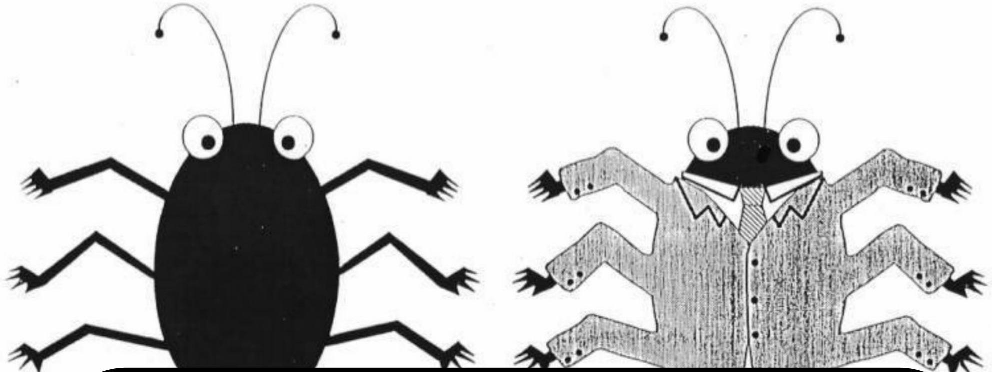
Terminology: bug vs. features

- what is a bug and what is a feature is subjective
- good rule of thumb: in any system with a large number of users, someone relies on every behavior of the system (intended or not) as if it were a feature



Terminology: bug vs. features

- what is a bug and what is a feature is **subjective**
- good rule of thumb: in any system with a large number of users, **someone** relies on every behavior of the system (intended or not) as if it were a feature



This is often why "**old**" systems (e.g., Linux, Windows, etc.) have behaviors that are **unintuitive** or difficult to learn: **someone relies on them**, so changing them would be considered a bug!

Debugging (Part 1/2)

Today's agenda:

- What is a bug, anyway?
- **Bug reports, triage, and the defect lifecycle**
- Debugging
  - printf debugging and logging
  - delta debugging
  - debuggers

Defect report lifecycle

Defect report lifecycle

Definition: the ***defect report lifecycle*** consists of a number of possible stages and actions, including reporting, confirmation, triage, assignment, resolution, and verification.

Defect report lifecycle

Definition: the ***defect report lifecycle*** consists of a number of possible stages and actions, including reporting, confirmation, triage, assignment, resolution, and verification.

- Not every defect report follows the same path

Defect report lifecycle

Definition: the ***defect report lifecycle*** consists of a number of possible stages and actions, including reporting, confirmation, triage, assignment, resolution, and verification.

- Not every defect report follows the same path
- The overall process is **not linear**
  - There are multiple entry points, some cycles, and multiple exit points (and some never leave …)

Defect report lifecycle

Definition: the ***defect report lifecycle*** consists of a number of possible stages and actions, including reporting, confirmation, triage, assignment, resolution, and verification.

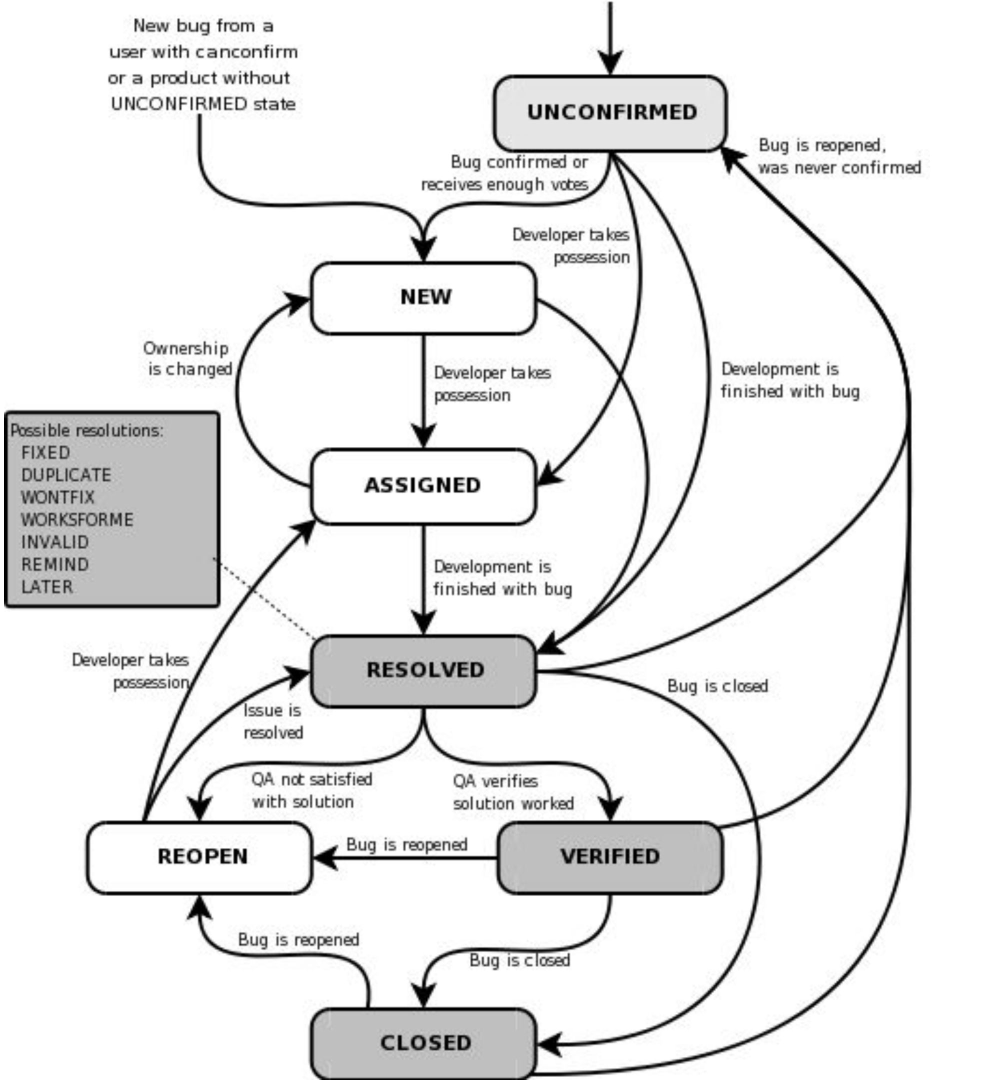
- Not every defect report follows the same path
- The overall process is **not linear**
  - There are multiple entry points, some cycles, and multiple exit points (and some never leave …)

Definition: the ***status*** of a defect report tracks its position in the lifecycle ("new", "resolved", etc.)

Defect report lifecycle

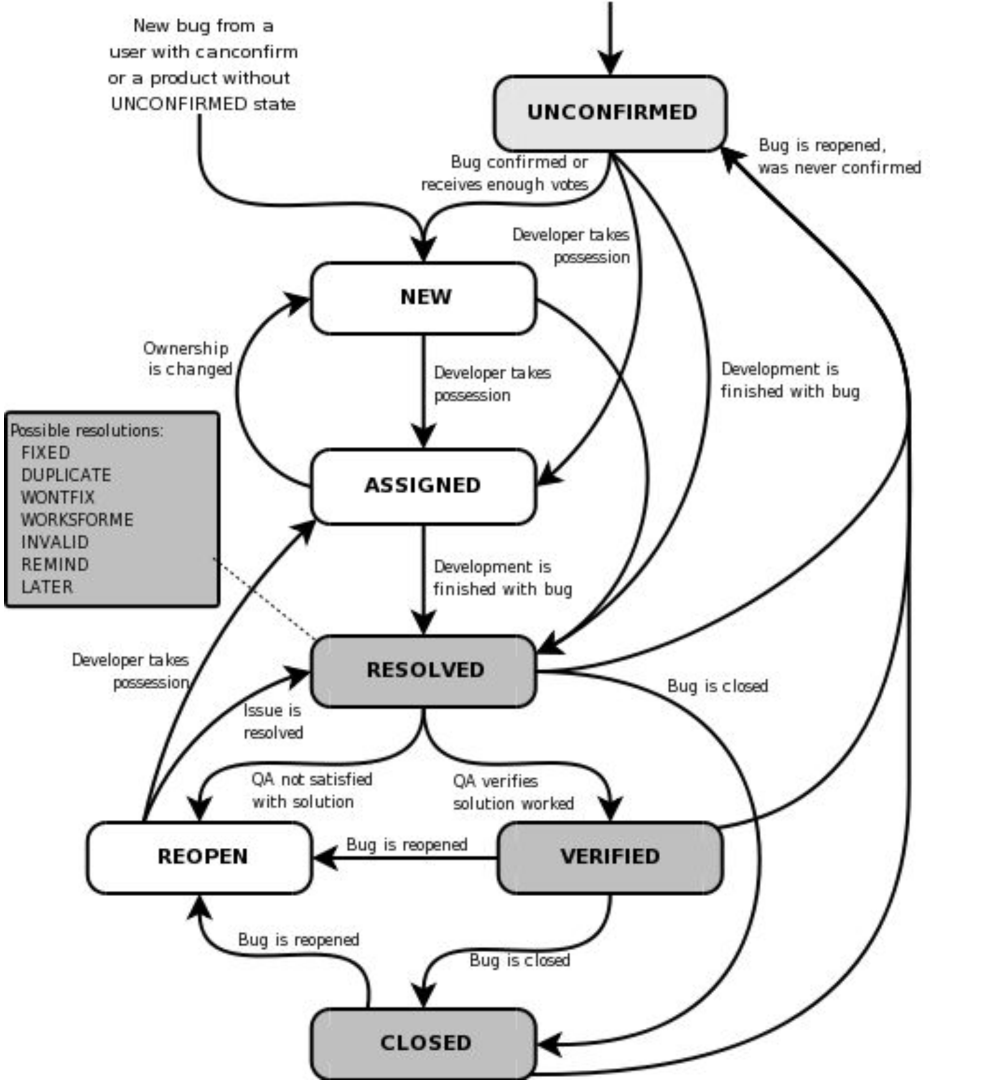
Defect report lifecycle

- For example, Bugzilla (a widely-used open-source issue tracker) uses **this** → flow for issues



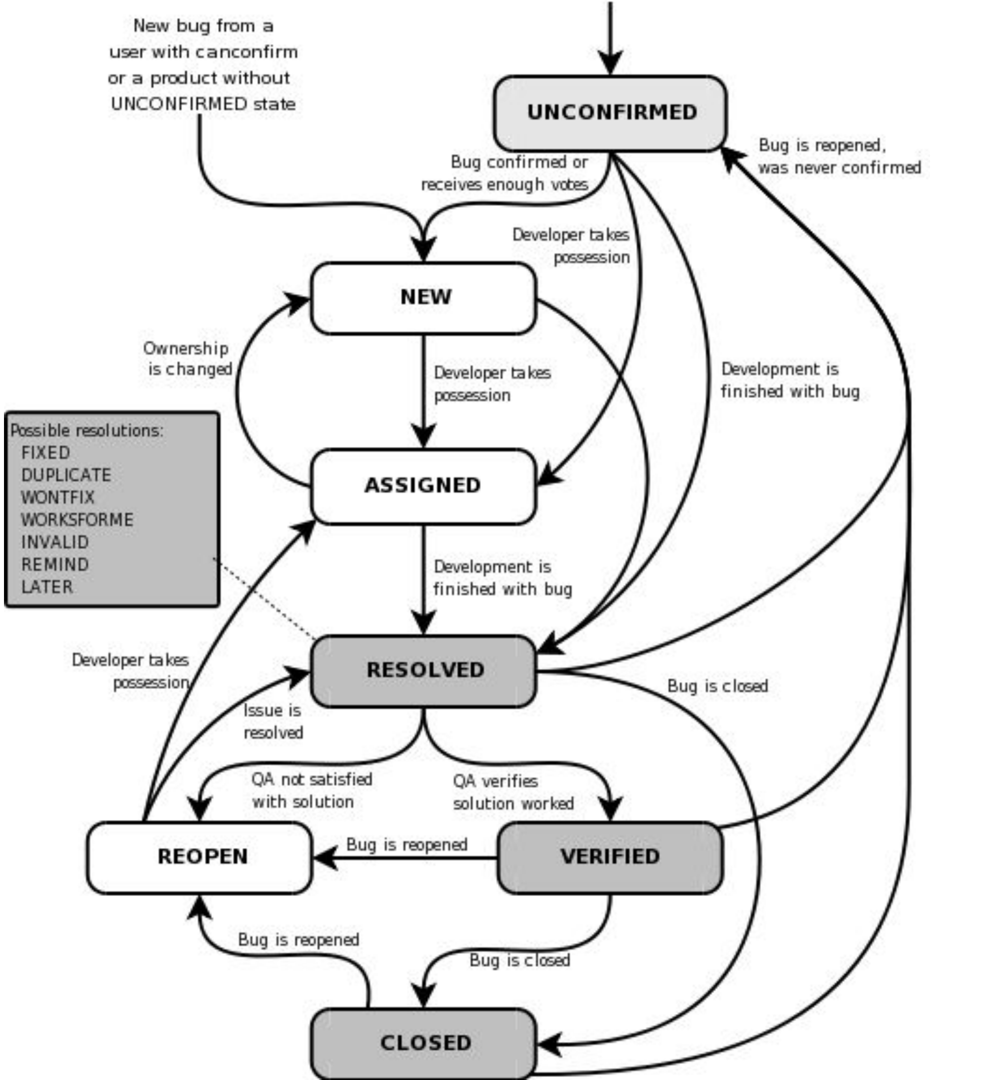
Defect report lifecycle

- For example, Bugzilla (a widely-used open-source issue tracker) uses **this** → flow for issues
- GitHub's built-in issue tracker is similar (less structured)



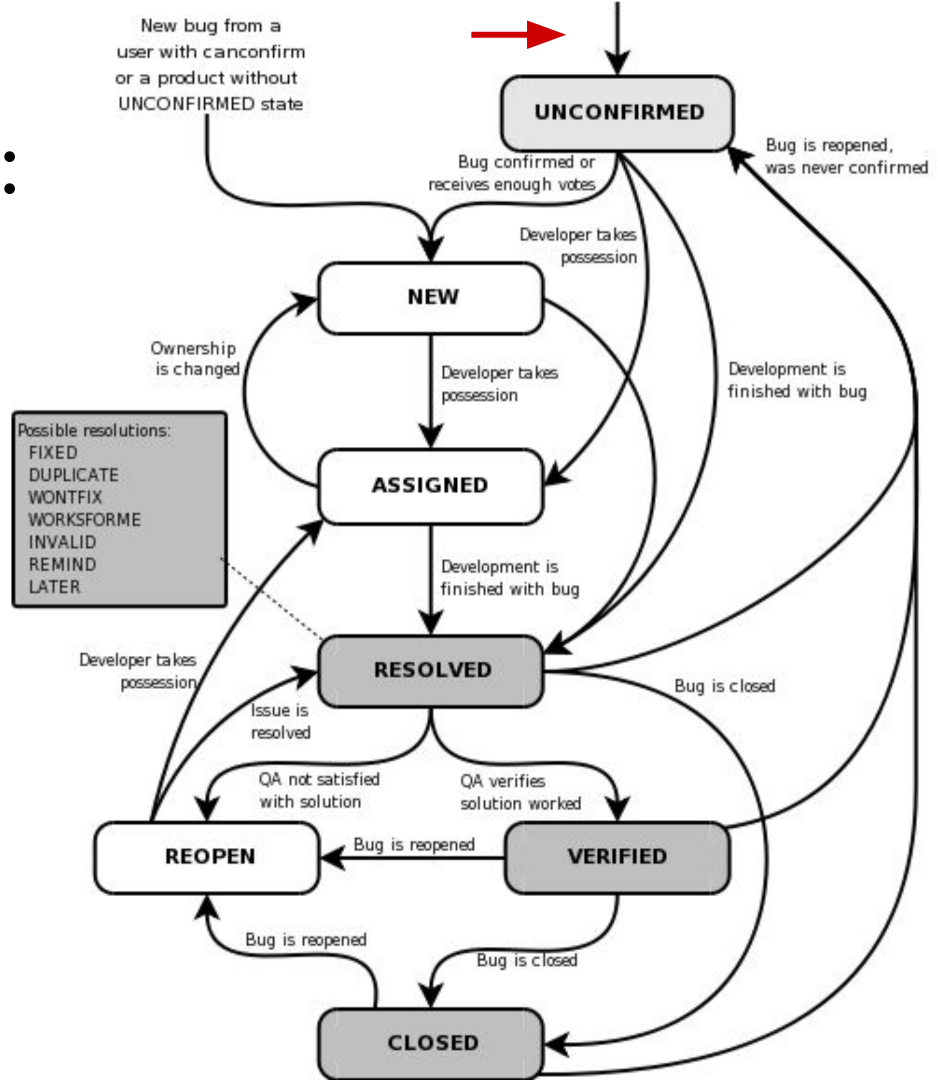
Defect report lifecycle

- For example, Bugzilla (a widely-used open-source issue tracker) uses **this** ⟶ flow for issues
- GitHub's built-in issue tracker is similar (less structured)
  - you should use an issue tracker for the group project (GitHub is okay)



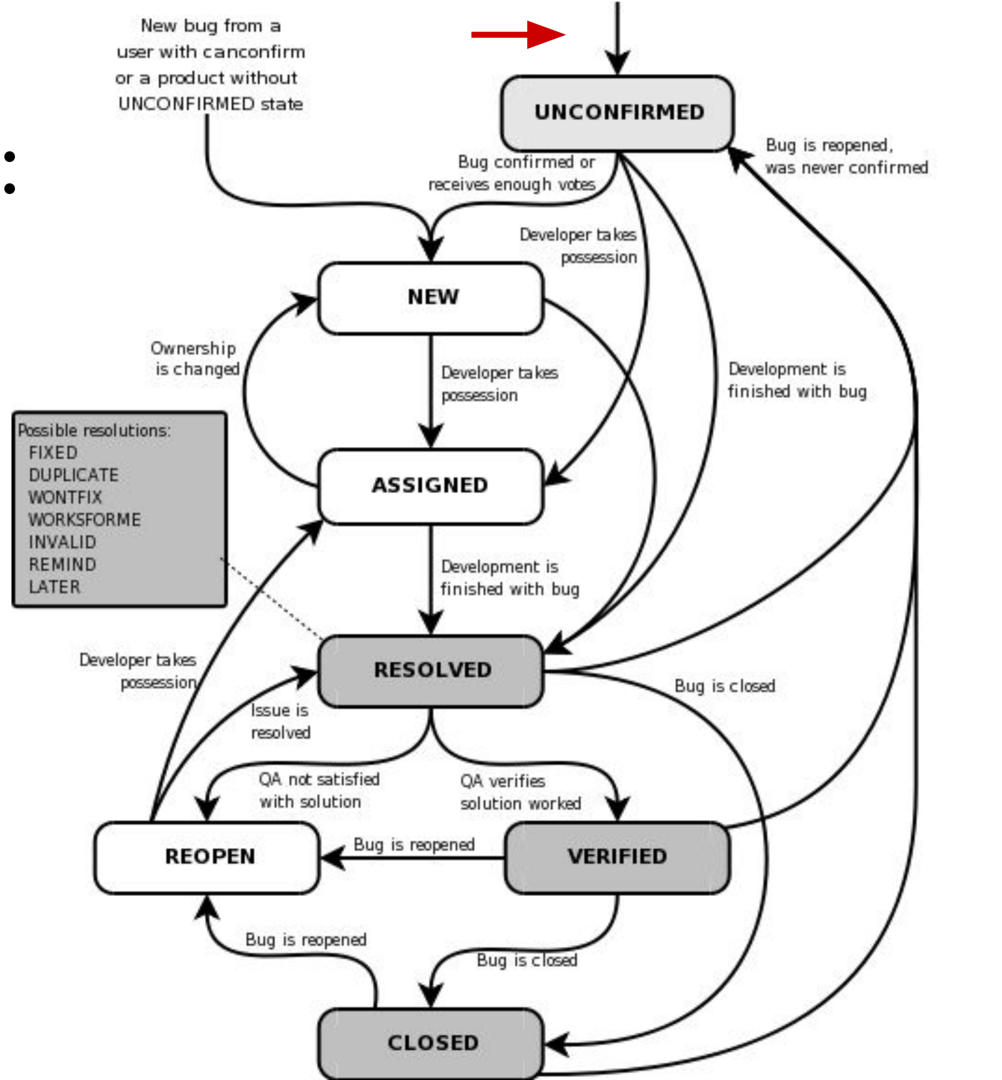
Defect report lifecycle: new bugs

- most new bugs enter the system as “unconfirmed”



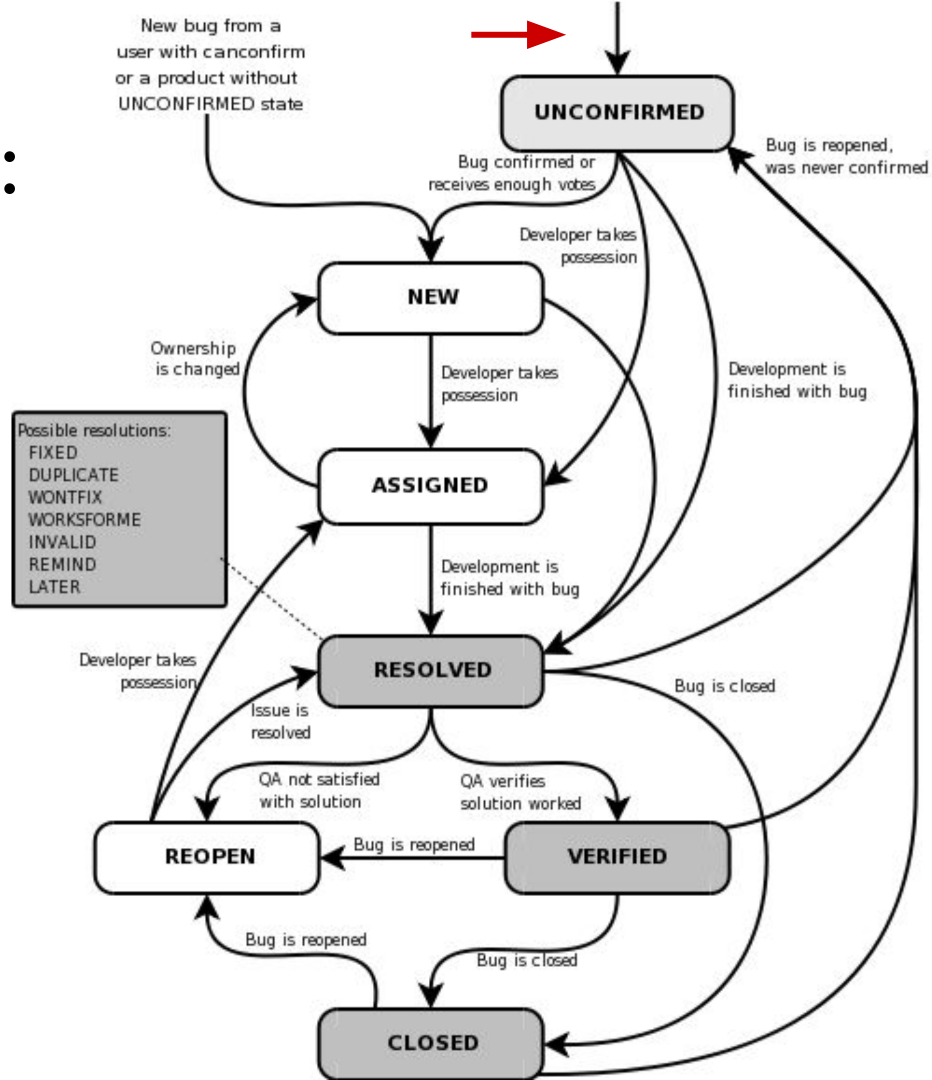
Defect report lifecycle: new bugs

- most new bugs enter the system as "unconfirmed"
- two main sources:



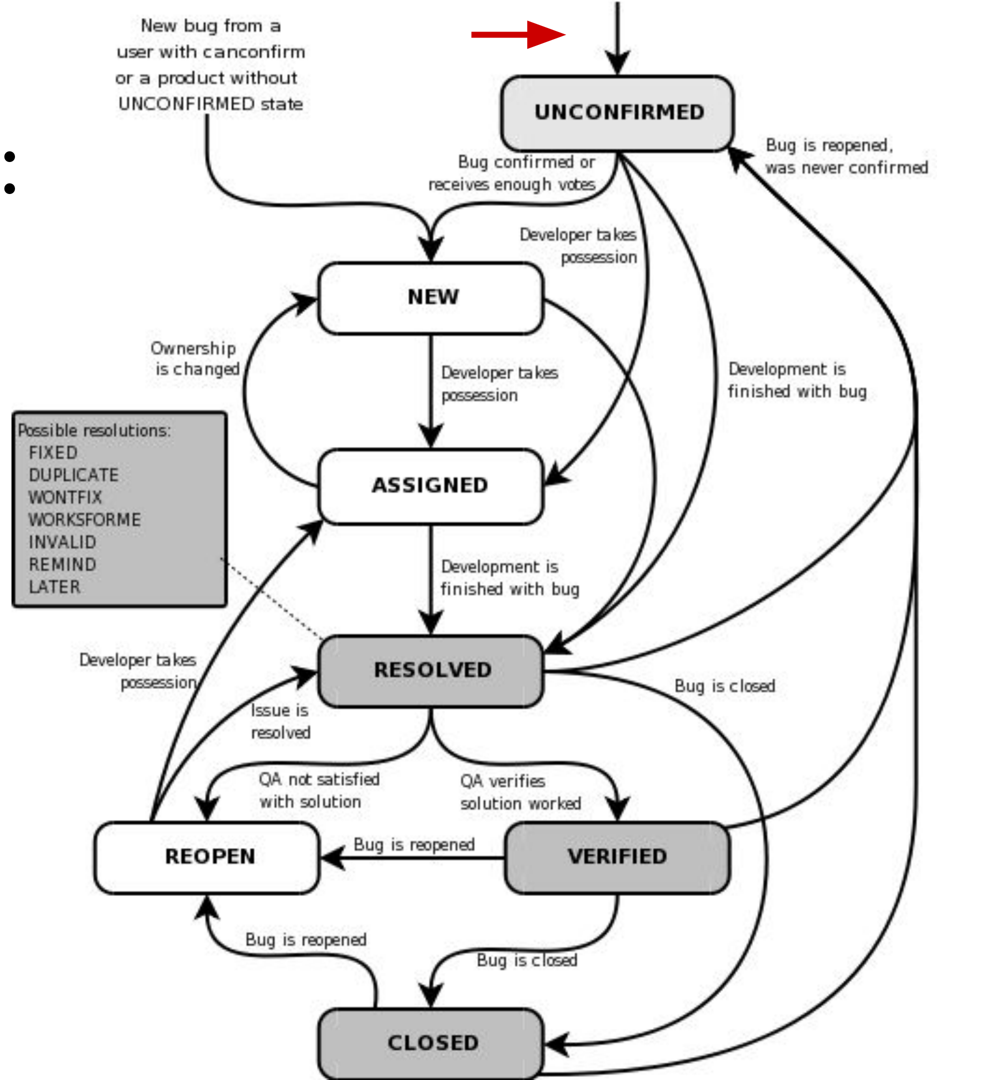
Defect report lifecycle: new bugs

- most new bugs enter the system as "unconfirmed"
- two main sources:
  - internal bug reports, e.g., from testers/QA



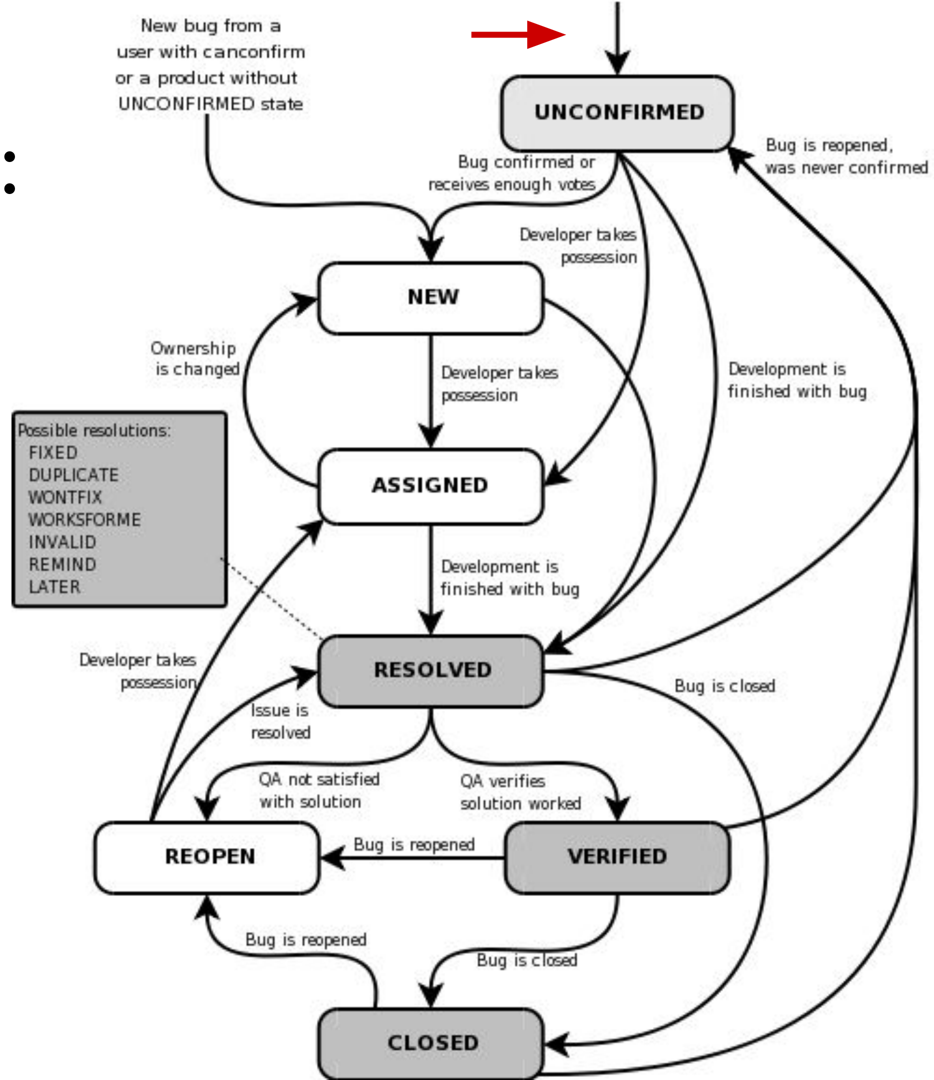
Defect report lifecycle: new bugs

- most new bugs enter the system as “unconfirmed”
- two main sources:
  - internal bug reports, e.g., from testers/QA
  - external bug reports, e.g., from users



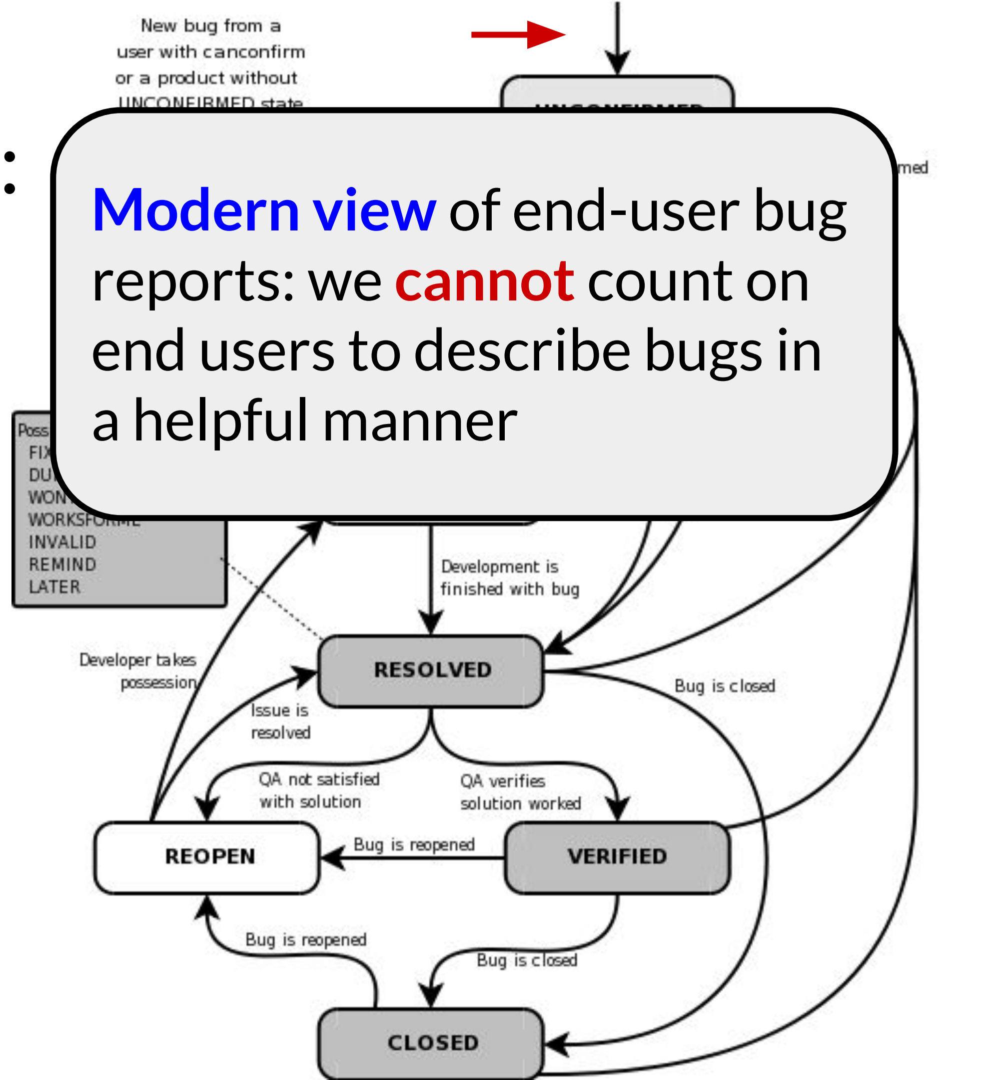
Defect report lifecycle: new bugs

- most new bugs enter the system as "unconfirmed"
- two main sources:
  - internal bug reports, e.g., from testers/QA
  - external bug reports, e.g., from users
- internal reports are *usually* higher quality/more detailed



Defect report lifecycle: new bugs

- most new bugs enter the system as “unconfirmed”
- two main sources:
  - internal bug reports, e.g., from testers/QA
  - external bug reports, e.g., from users
- internal reports are *usually* higher quality/more detailed

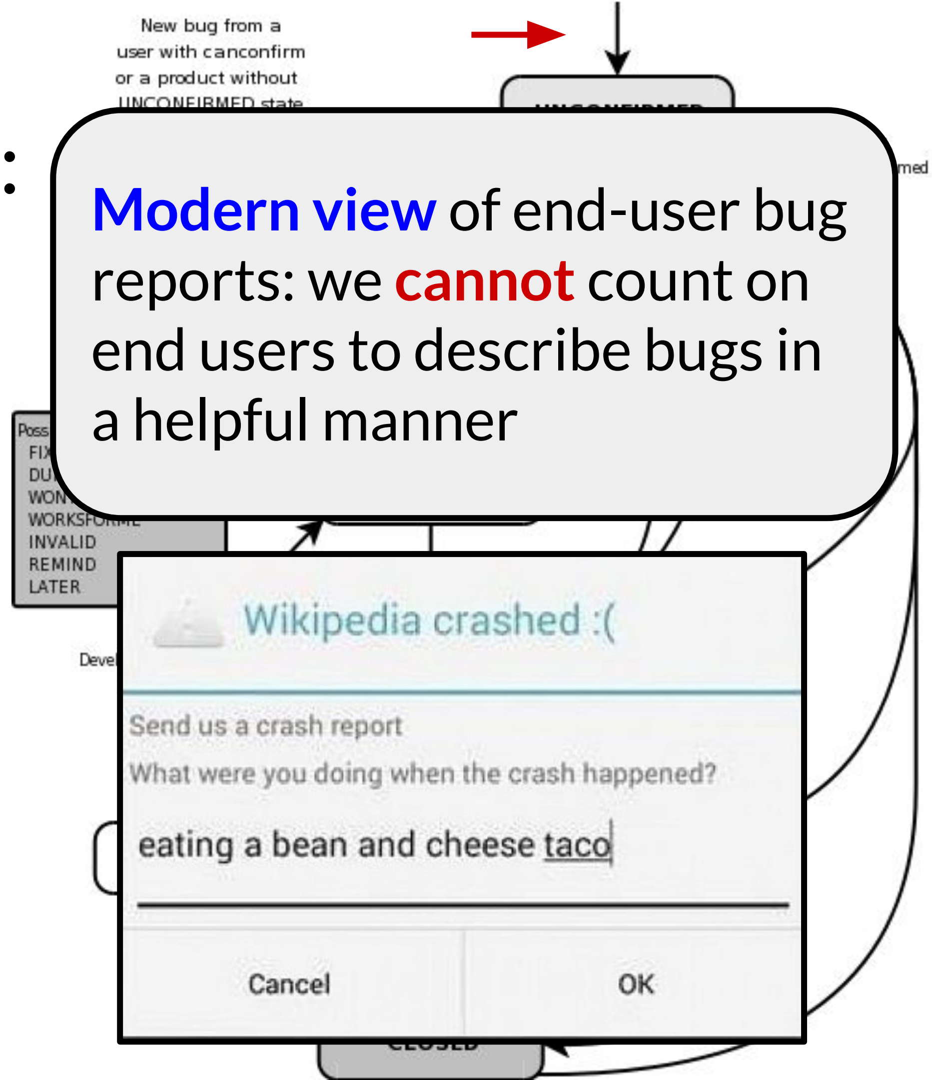


Modern view of end-user bug reports: we **cannot** count on end users to describe bugs in a helpful manner

Defect report lifecycle: new bugs

- most new bugs enter the system as “unconfirmed”
- two main sources:
  - internal bug reports, e.g., from testers/QA
  - external bug reports, e.g., from users
- internal reports are *usually* higher quality/more detailed

Modern view of end-user bug reports: we **cannot** count on end users to describe bugs in a helpful manner



Quick demo: GitHub issue tracker

example: https://github.com/typetools/checker-framework/issues

Writing a good defect report

- clearly explain:

Writing a good defect report

- clearly explain:
    - what **you did**
        - ideally, by providing a set of commands that can be pasted into a shell and reproduce the problem

Writing a good defect report

- clearly explain:
    - what **you did**
        - ideally, by providing a set of commands that can be pasted into a shell and reproduce the problem
    - what the **program did**
        - usually you should copy-paste output, but this could also be screenshots, video, etc.

Writing a good defect report

- clearly explain:
  - what **you did**
    - ideally, by providing a set of commands that can be pasted into a shell and reproduce the problem
  - what the **program did**
    - usually you should copy-paste output, but this could also be screenshots, video, etc.
  - **why** you believe that what the program did is wrong

Writing a good defect report

- clearly explain:
  - what **you did**
    - ideally, by providing a set of commands that can be pasted into a shell and reproduce the problem
  - what the **program did**
    - usually you should copy-paste output, but this could also be screenshots, video, etc.
  - **why** you believe that what the program did is wrong
  - what you **expected** the program to do instead

Defect reports: conversations

Defect reports: conversations

- Defect reports are **not static**

Defect reports: conversations

- Defect reports are **not static**
- Instead, they are **updated over time**
  - Request more info
  - Assign to a dev
  - Discuss solutions

Defect reports: conversations

- Defect reports are **not static**
- Instead, they are **updated over time**
  - Request more info
  - Assign to a dev
  - Discuss solutions
- The report is a **log** of all relevant activity

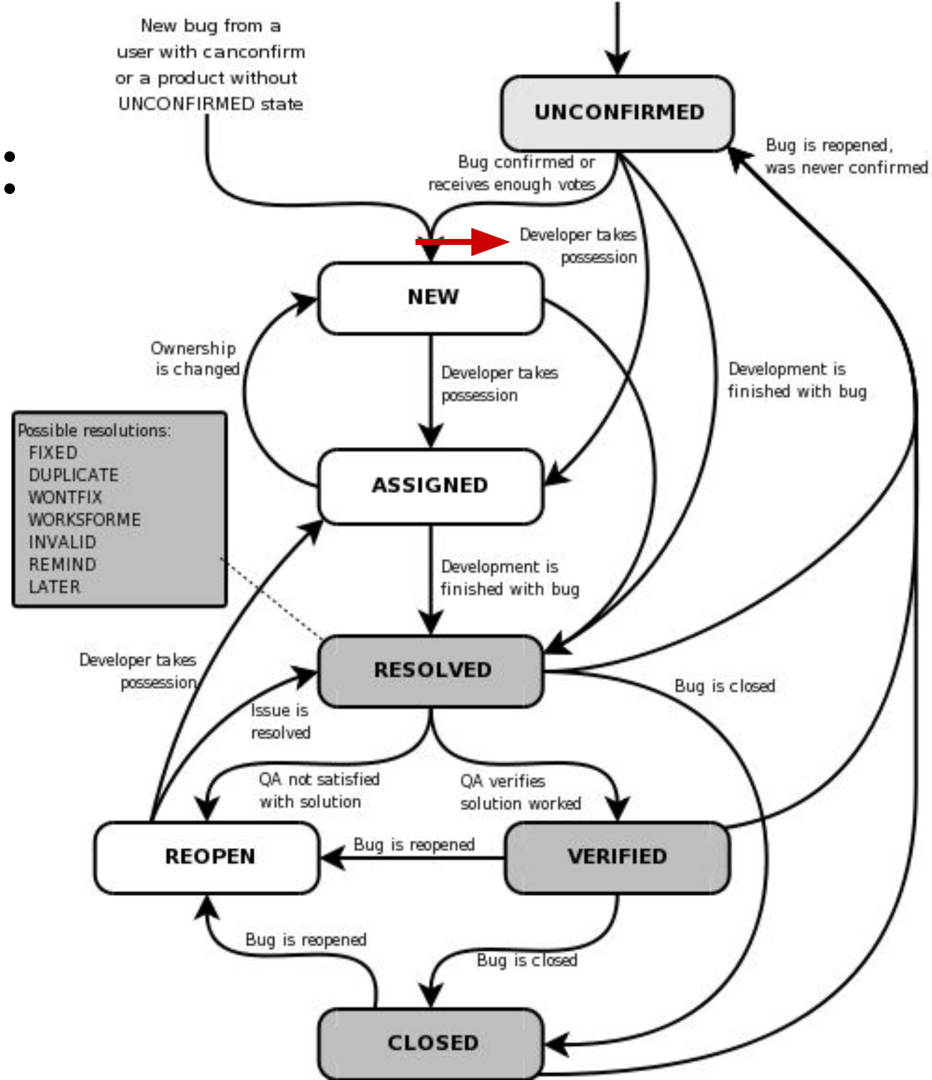
Defect reports: conversations

- Defect reports are **not static**
- Instead, they are **updated over time**
  - Request more info
  - Assign to a dev
  - Discuss solutions
- The report is a **log** of all relevant activity
- e.g.:
  - https://github.com/typetools/checker-framework/issues/4838

Defect reports: conversations

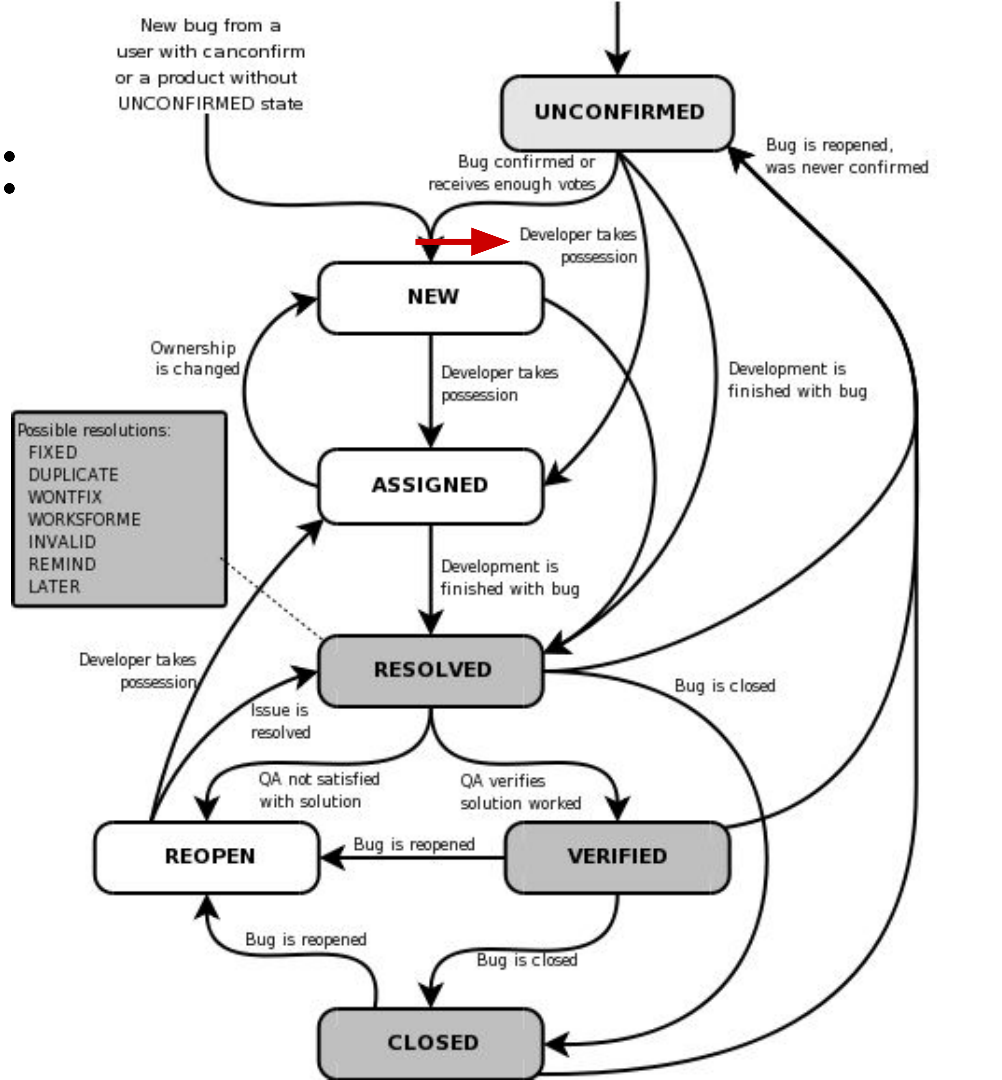
- Defect reports are **not static**
- Instead, they are **updated over time**
  - Request more info
  - Assign to a dev
  - Discuss solutions
- The report is a **log** of all relevant activity
- e.g.:
  - https://github.com/typetools/checker-framework/issues/4838
  - https://github.com/typetools/checker-framework/issues/3001

Defect report lifecycle: triage



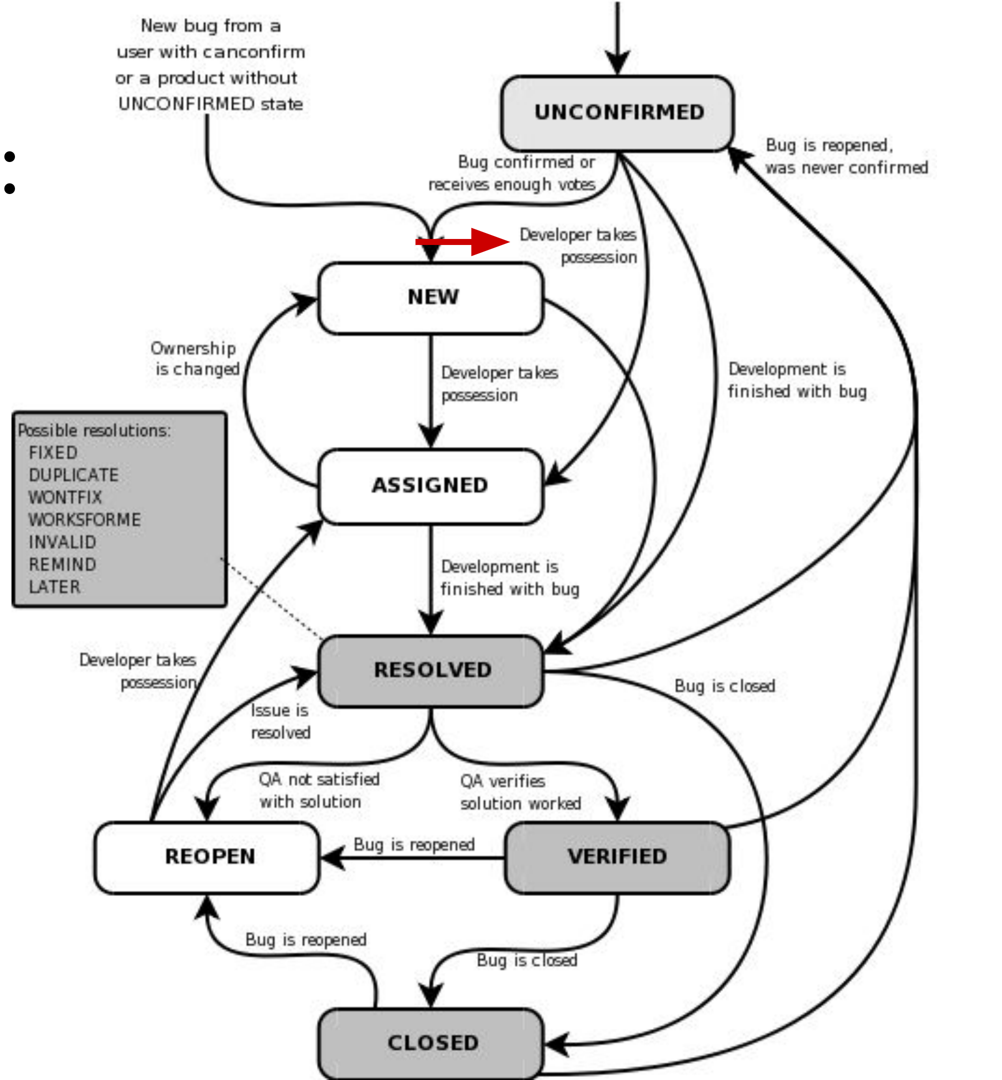
Defect report lifecycle: triage

- Key question: **which** bugs should we address first?



Defect report lifecycle: triage

- Key question: **which** bugs should we address first?
- "***triage***" is an analogy to **medicine**: which emergency room patient should you help first?



Defect report lifecycle: triage

Definition: ***triage*** is the assignment of degrees of urgency to wounds or illnesses to decide the order of treatment of a large number of patients or casualties

Defect report lifecycle: triage

Definition: ***triage*** is the assignment of degrees of urgency to wounds or illnesses to decide the order of treatment of a large number of patients or casualties

- ***bug triage*** has the same definition, but with software defects instead of wounds/illnesses

Defect report lifecycle: triage

Definition: ***triage*** is the assignment of degrees of urgency to wounds or illnesses to decide the order of treatment of a large number of patients or casualties

- ***bug triage*** has the same definition, but with software defects instead of wounds/illnesses
- there are always **more defect reports than resources** available to address them

Defect report lifecycle: triage

Definition: ***triage*** is the assignment of degrees of urgency to wounds or illnesses to decide the order of treatment of a large number of patients or casualties

- ***bug triage*** has the same definition, but with software defects instead of wounds/illnesses
- there are always **more defect reports than resources** available to address them
- we must do **cost-benefit** analysis:
  - How expensive is it to **fix** this bug?
  - How expensive is it to **not fix** this bug?

Defect report lifecycle: severity

Definition: ***severity*** is the degree of impact that a defect has on the development or operation of a component or system

Defect report lifecycle: severity

Definition: ***severity*** is the degree of impact that a defect has on the development or operation of a component or system

- intuition: severity = "cost of **not fixing** the bug"

Defect report lifecycle: severity

Definition: ***severity*** is the degree of impact that a defect has on the development or operation of a component or system

- intuition: severity = "cost of **not fixing** the bug"
- BugZilla severity levels (varies by company/tool, but these typical):

Severity	Meaning
Blocker	Blocks further development and/or testing work.
Critical	Crashes, loss of data (internally, not your edit preview!) in a widely used and important component.
Major	Major loss of function in an important area.
Normal	Default/average.
Minor	Minor loss of function, or other problem that does not affect many people or where an easy workaround is present.
Trivial	Cosmetic problem like misspelled words or misaligned text which does not really cause problems.
Enhancement	Request for a new feature or change in functionality for an existing feature.

Defect report lifecycle: priority

Definition: ***priority*** indicates the importance or urgency of fixing a defect

Defect report lifecycle: priority

Definition: ***priority*** indicates the importance or urgency of fixing a defect

- related to, but officially different from, severity
  - **intuition**: if you have lots of high severity bugs, you need to prioritize between them

Defect report lifecycle: pr

Definition: ***priority*** indicates the im
defect

- related to, but officially differe
  - **intuition**: if you have lots of
    prioritize between them

Usually, "**high priority**" = "a developer will work on this soon" (e.g., in the next sprint).

Defect report lifecycle: pri...

Definition: ***priority*** indicates the im...
defect

- related to, but officially differe...
  - **intuition**: if you have lots of...
    prioritize between them

Usually, "**high priority**" = "a developer will work on this soon" (e.g., in the next sprint).

"As a rule of thumb, limit High priority task assignments for a single person to three, five in exceptional times."

Defect report lifecycle: priority

Definition: ***priority*** indicates the importance or urgency of fixing a defect

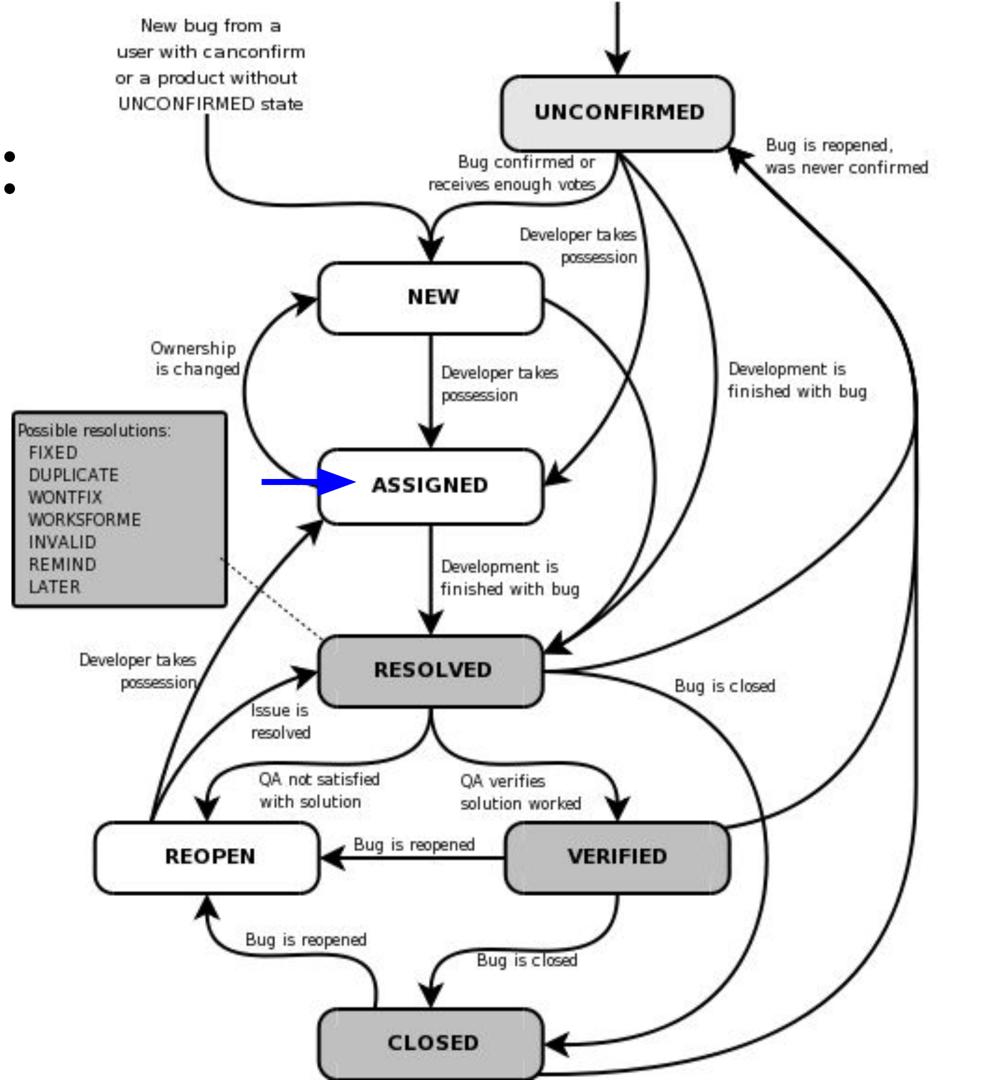
- related to, but officially different from, severity
  - **intuition**: if you have lots of high severity bugs, you need to prioritize between them
- severity and priority are used **together** (along with complexity, risk, etc.) to evaluate, prioritize and assign the resolution of reports

Defect report lifecycle: priority

Definition: ***priority*** indicates the importance or urgency of fixing a defect

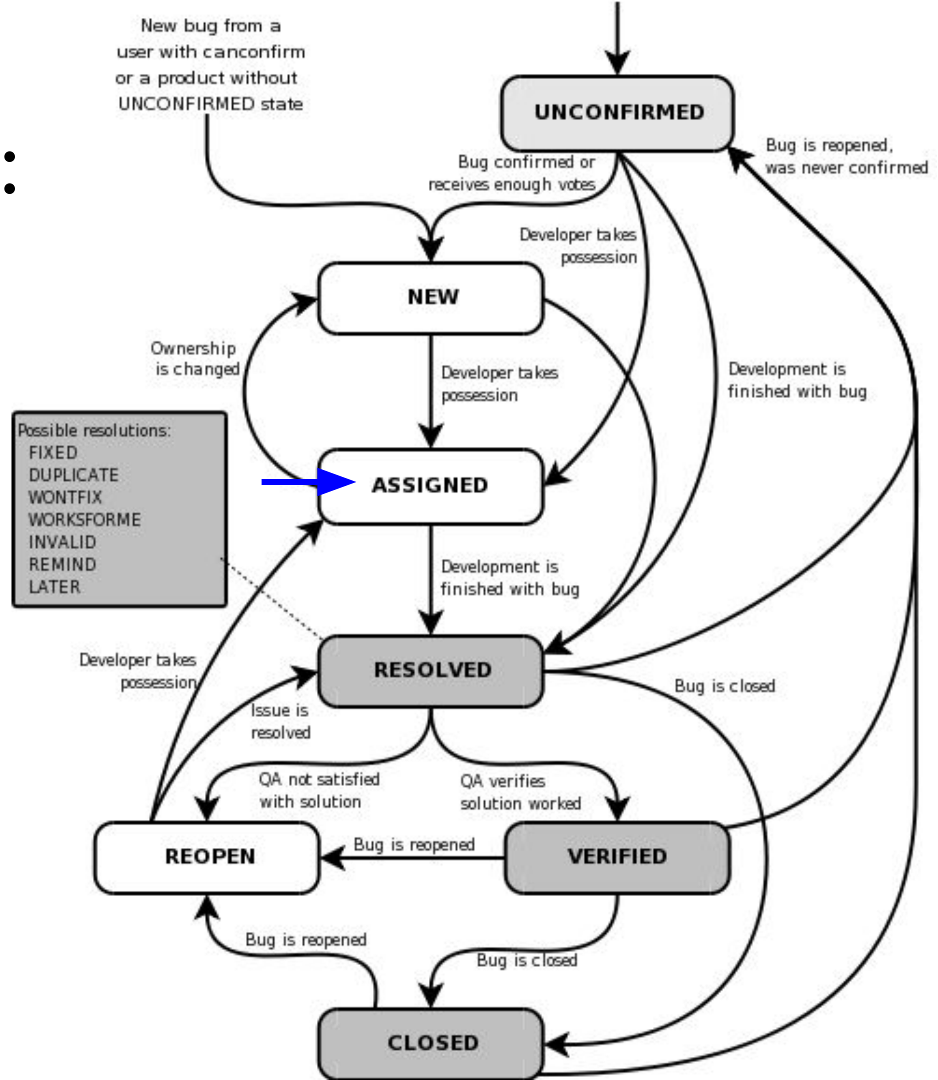
- related to, but officially different from, severity
  - **intuition**: if you have lots of high severity bugs, you need to prioritize between them
- severity and priority are used **together** (along with complexity, risk, etc.) to evaluate, prioritize and assign the resolution of reports
  - note that this is a bit of an **oversimplification**:
    "severity + priority = triage" is like "supply + demand = price"

Defect report lifecycle: assignment



Defect report lifecycle: assignment

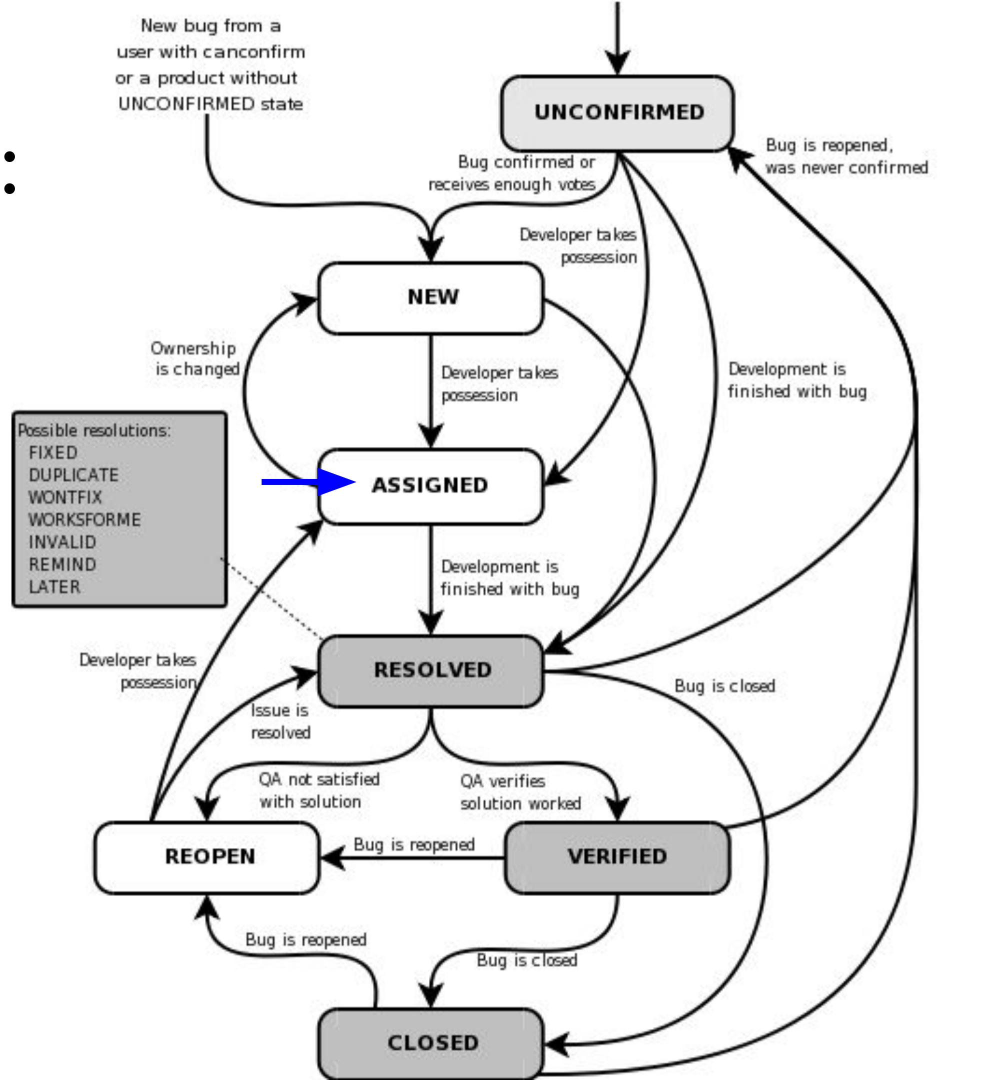
- Key question: **who** should fix this bug?



Defect report lifecycle: assignment

- Key question: **who** should fix this bug?

Definition: an ***assignment*** associates a developer with the responsibility of addressing a defect report

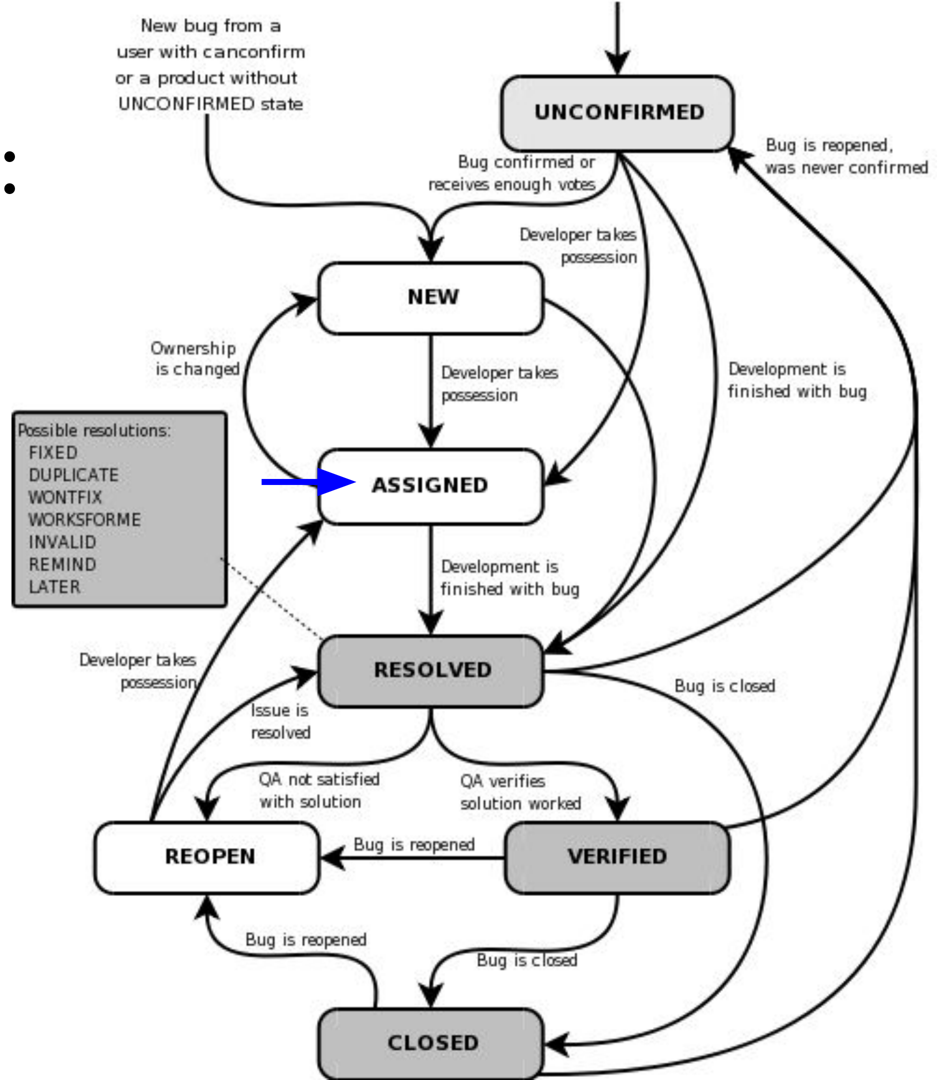


Defect report lifecycle: assignment

- Key question: **who** should fix this bug?

Definition: an ***assignment*** associates a developer with the responsibility of addressing a defect report

- state of the art is "manual"

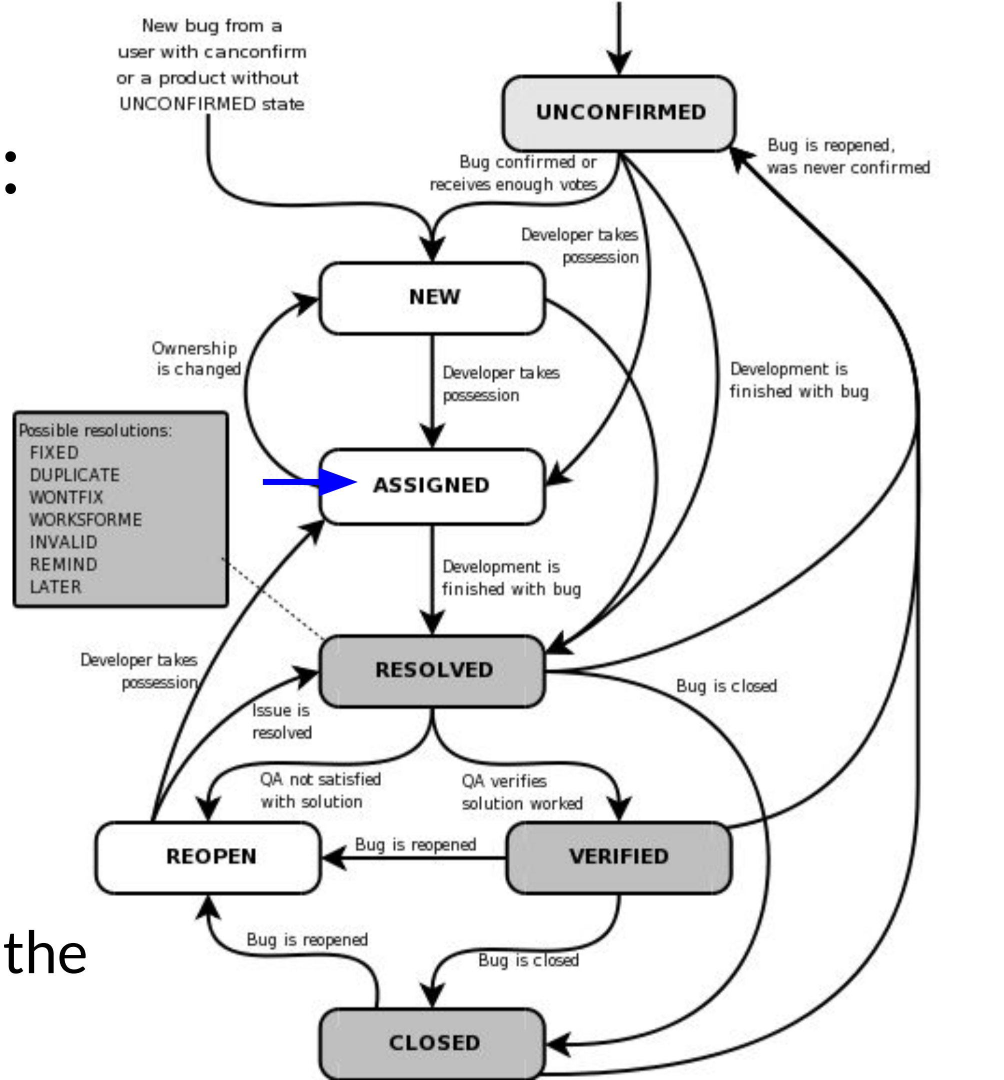


Defect report lifecycle: assignment

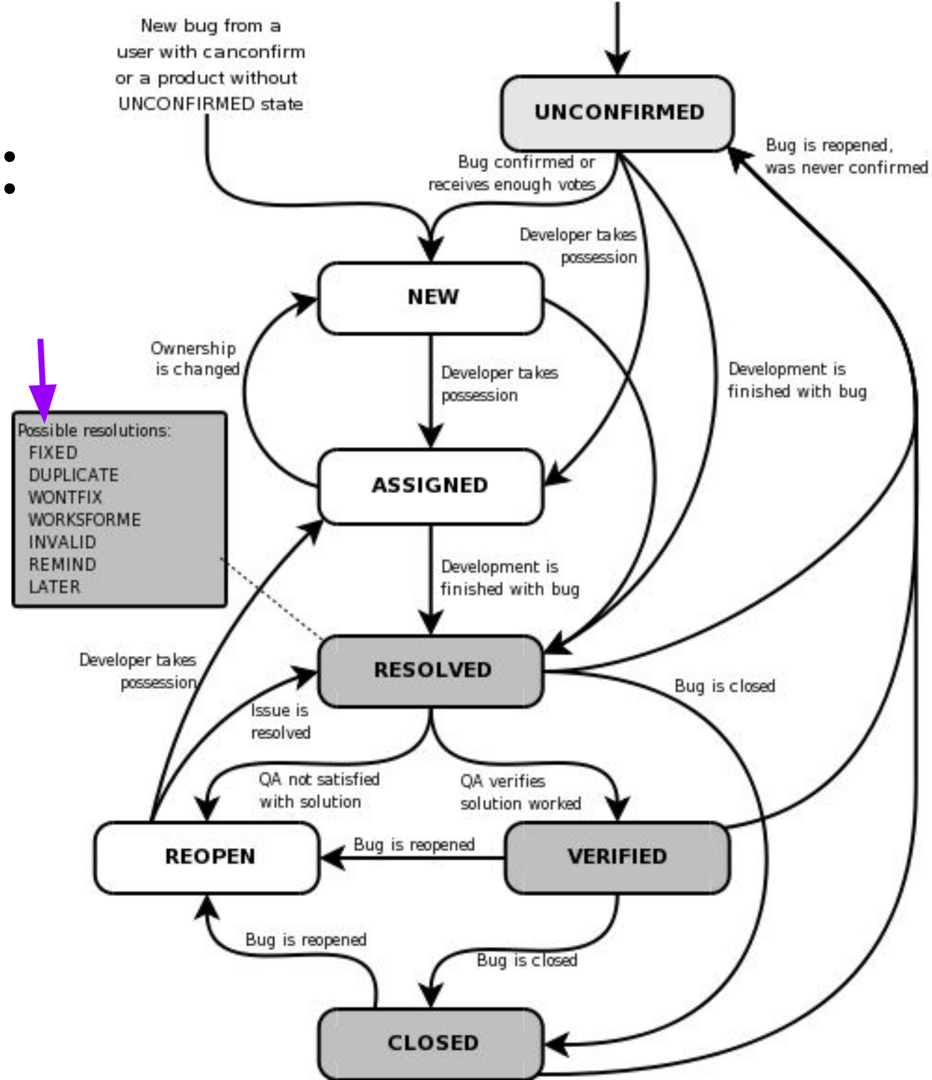
- Key question: **who** should fix this bug?

Definition: an ***assignment*** associates a developer with the responsibility of addressing a defect report

- state of the art is “manual”
- usually based on who “owns” the relevant code

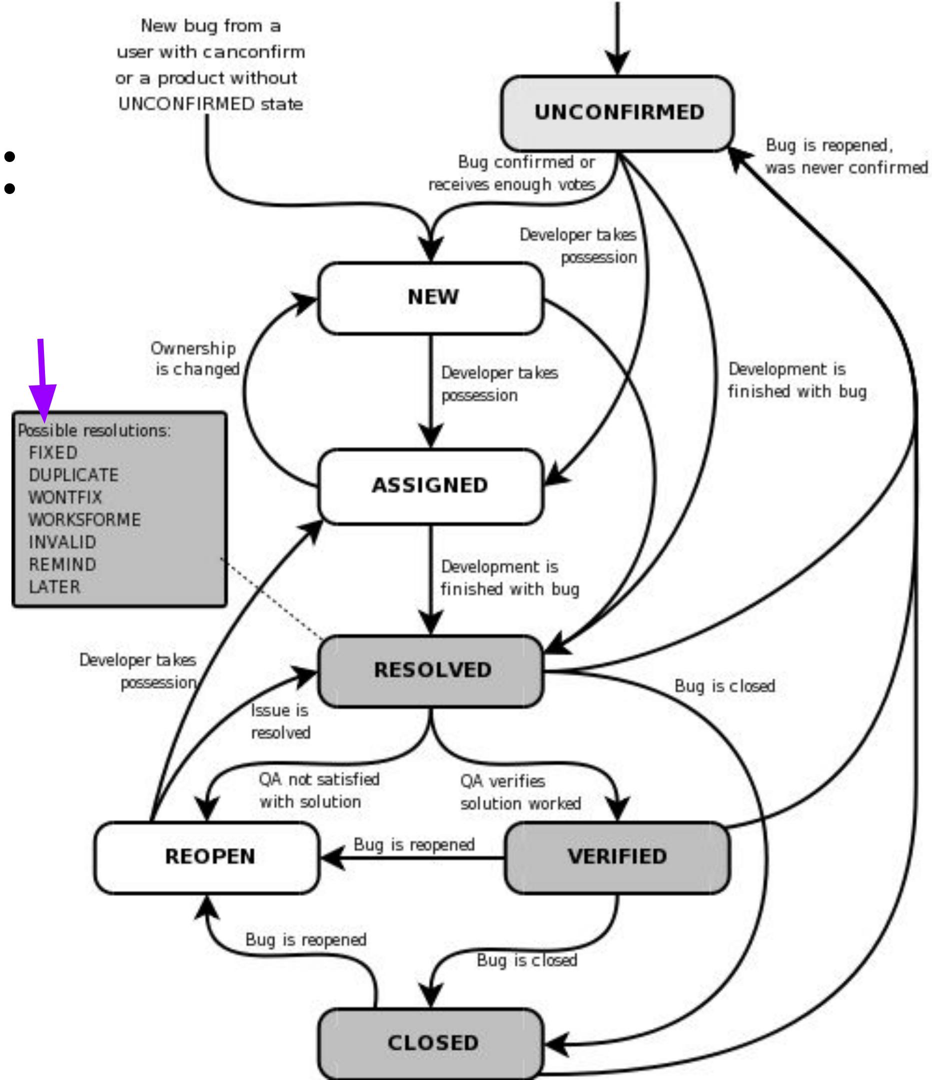


Defect report lifecycle: resolution



Defect report lifecycle: resolution

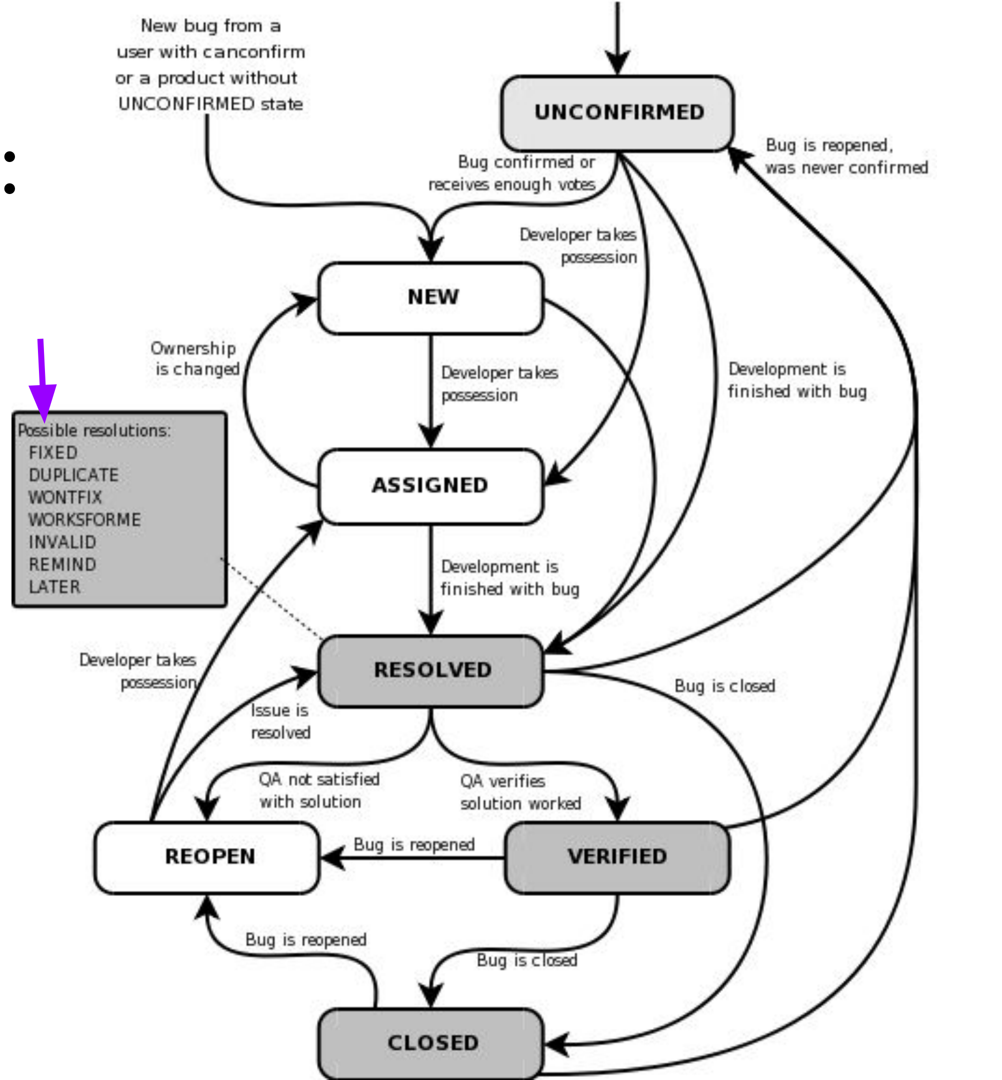
- Key question: did we **fix** it?



Defect report lifecycle: resolution

- Key question: did we **fix** it?

Definition: a defect report ***resolution*** status indicates the result of the most recent attempt to address it

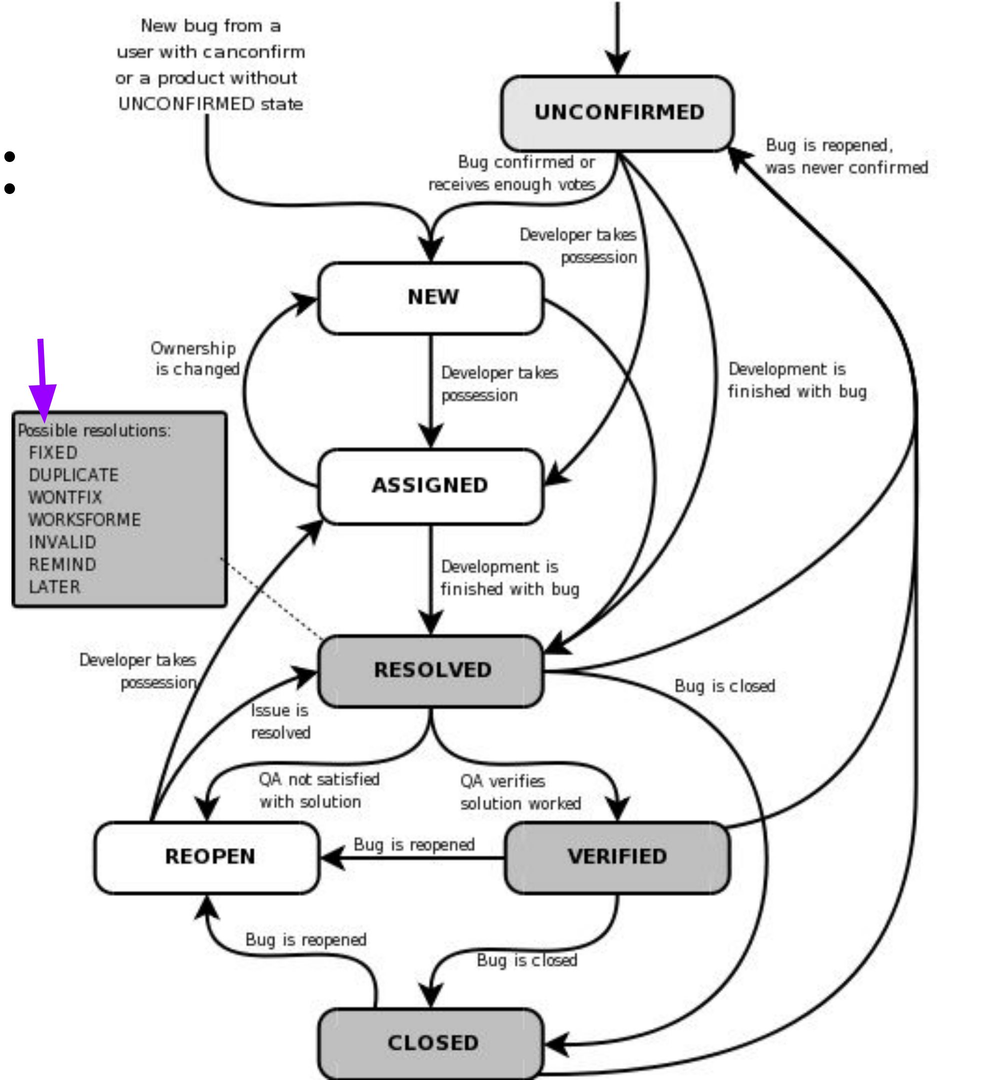


Defect report lifecycle: resolution

- Key question: did we **fix** it?

Definition: a defect report ***resolution*** status indicates the result of the most recent attempt to address it

- **Important**: resolved **need not** mean "fixed"



Defect report lifecycle: possible resolutions

BugZilla resolution options:

- **FIXED** (give commit #)

Defect report lifecycle: possible resolutions

BugZilla resolution options:

- **FIXED** (give commit #)
- **INVALID** (bug report is invalid)
- **WONTFIX** (we don't ever plan to fix it)
- **DUPLICATE** (link to other bug report #)
- **WORKSFORME** (cannot reproduce, a.k.a. "WFM")
- **MOVED** (give link: filed with wrong project)
- **NOTABUG** (report describes expected behavior)
- **NOTOURBUG** (is a bug, but not with our software)
- **INSUFFICIENTDATA** (cannot triage/fix w/o more)

Defect report lifecycle: possible resolutions

BugZilla resolution options:

- **FIXED** (give commit #)
- **INVALID** (bug report is invalid)
- **WONTFIX** (we don't ever plan to fix it)
- **DUPLICATE** (link to other bug report #)
- **WORKSFORME** (cannot reproduce, a.k.a. “WFM”)
- **MOVED** (give link: filed with wrong project)
- **NOTABUG** (report describes expected behavior)
- **NOTOURBUG** (is a bug, but not with our software)
- **INSUFFICIENTDATA** (cannot triage/fix w/o more)

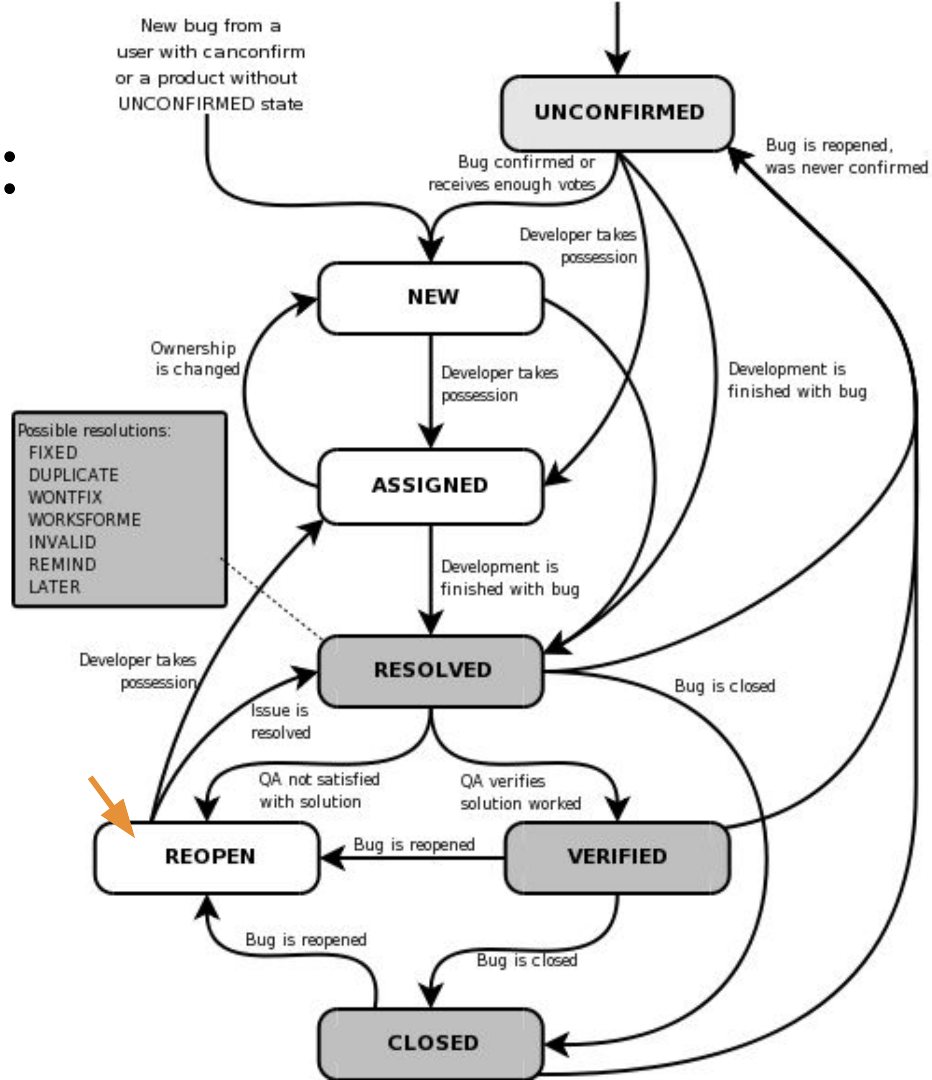
Thought question: what **fraction** of bug reports end up with each resolution?

Defect report lifecycle: possible resolutions

A significant fraction of submitted bug reports are spurious duplicates that describe already-reported defects. Previous studies report that as many as 36% of bug reports were duplicates or otherwise invalid [2]. Of the 29,000 bug reports used in the experiments in this paper, 25.9% were identified as duplicates by the project developers.

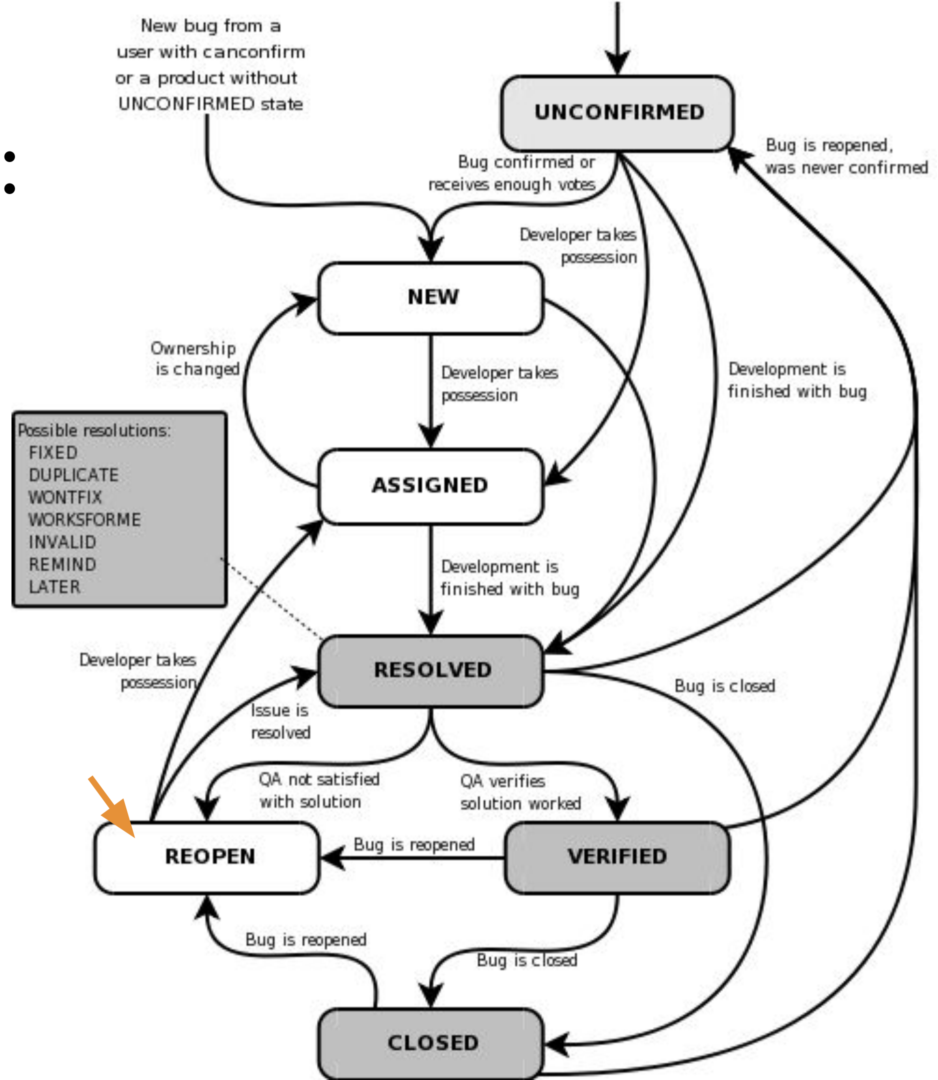
[Jalbert et al. Automated Duplicate Detection for Bug Tracking Systems. DSN 2008.]

Defect report lifecycle: reopening



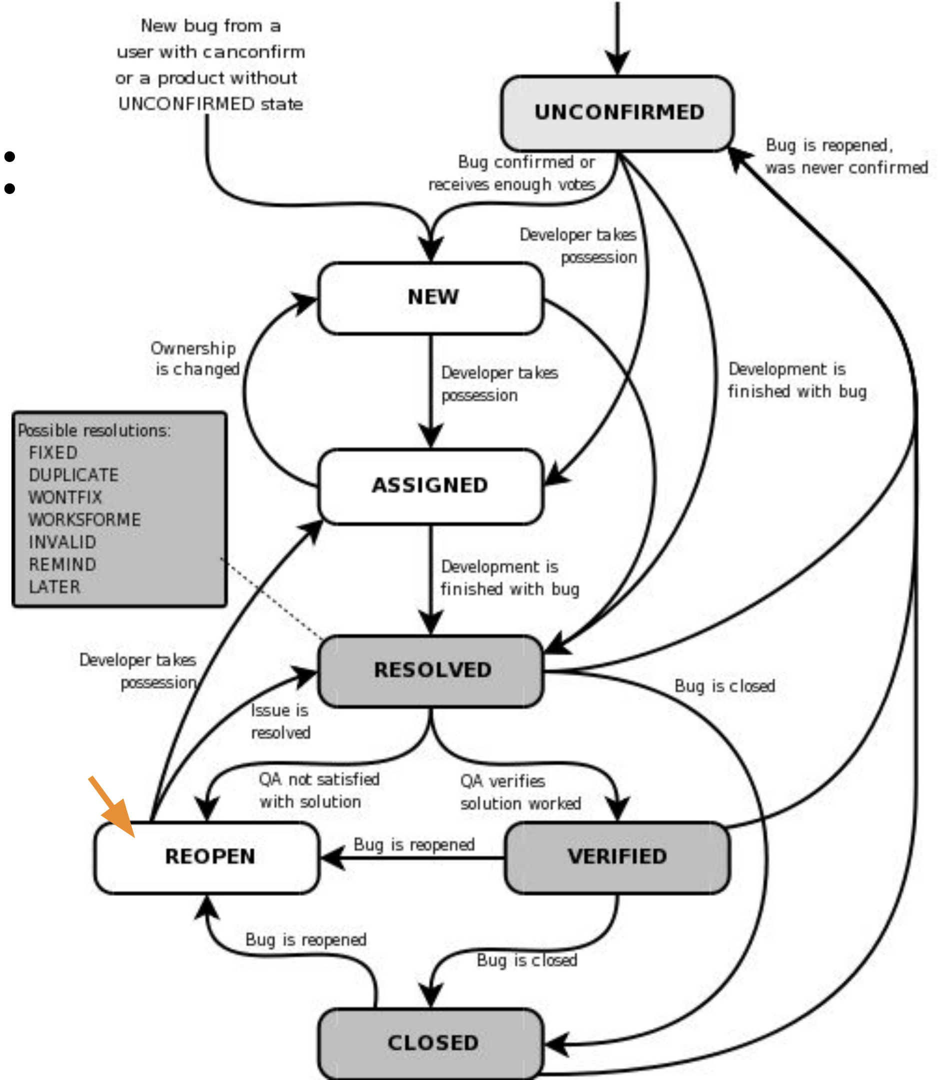
Defect report lifecycle: reopening

- A defect report that was previously resolved (e.g. “FIXED”) may be **reopened** if later evidence suggests the old resolution is no longer adequate



Defect report lifecycle: reopening

- A defect report that was previously resolved (e.g. “FIXED”) may be **reopened** if later evidence suggests the old resolution is no longer adequate
- Surely this only happens **rarely**?



Defect report lifecycle: reopening

This paper presents a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases including Linux, OpenSolaris, FreeBSD and also a mature *commercial* OS developed and evolved over the last *12* years, investigating not only the mistake patterns during bug-fixing but also the possible *human reasons* in the development process when these incorrect bug-fixes were introduced. Our major findings include: (1) at least 14.8%~24.4% of sampled fixes for post-release bugs [1] in these large OSes are incorrect and have made impacts to end users. (2) Among several common bug types, concurrency bugs are the most difficult to fix correctly: 39% of concurrency bug fixes are incorrect. (3) Developers and reviewers for incorrect fixes usually do not have enough knowledge about the involved code. For example, 27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix. Our results provide useful guidelines to design new tools and also to improve the development process. Based on our findings, the commercial software

- Many fixes are **wrong**, even on mature, critical software!

[Yin et al. How Do Fixes Become Bugs? ESEC/FSE 2011.]

Defect report lifecycle: reopening

> This paper presents a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases including Linux, OpenSolaris, FreeBSD and also a mature *commercial* OS developed and evolved over the last *12* years, investigating not only the mistake patterns during bug-fixing but also the possible *human reasons* in the development process when these incorrect bug-fixes were introduced. Our major findings include: (1) at least 14.8%~24.4% of sampled fixes for post-release bugs [1] in these large OSes are incorrect and have made impacts to end users. (2) Among several common bug types, concurrency bugs are the most difficult to fix correctly: 39% of concurrency bug fixes are incorrect. (3) Developers and reviewers for incorrect fixes usually do not have enough knowledge about the involved code. For example, 27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix. Our results provide useful guidelines to design new tools and also to improve the development process. Based on our findings, the commercial software

- Many fixes are **wrong**, even on mature, critical software!
- Implication: reopening bugs is **common**

[Yin et al. How Do Fixes Become Bugs? ESEC/FSE 2011.]

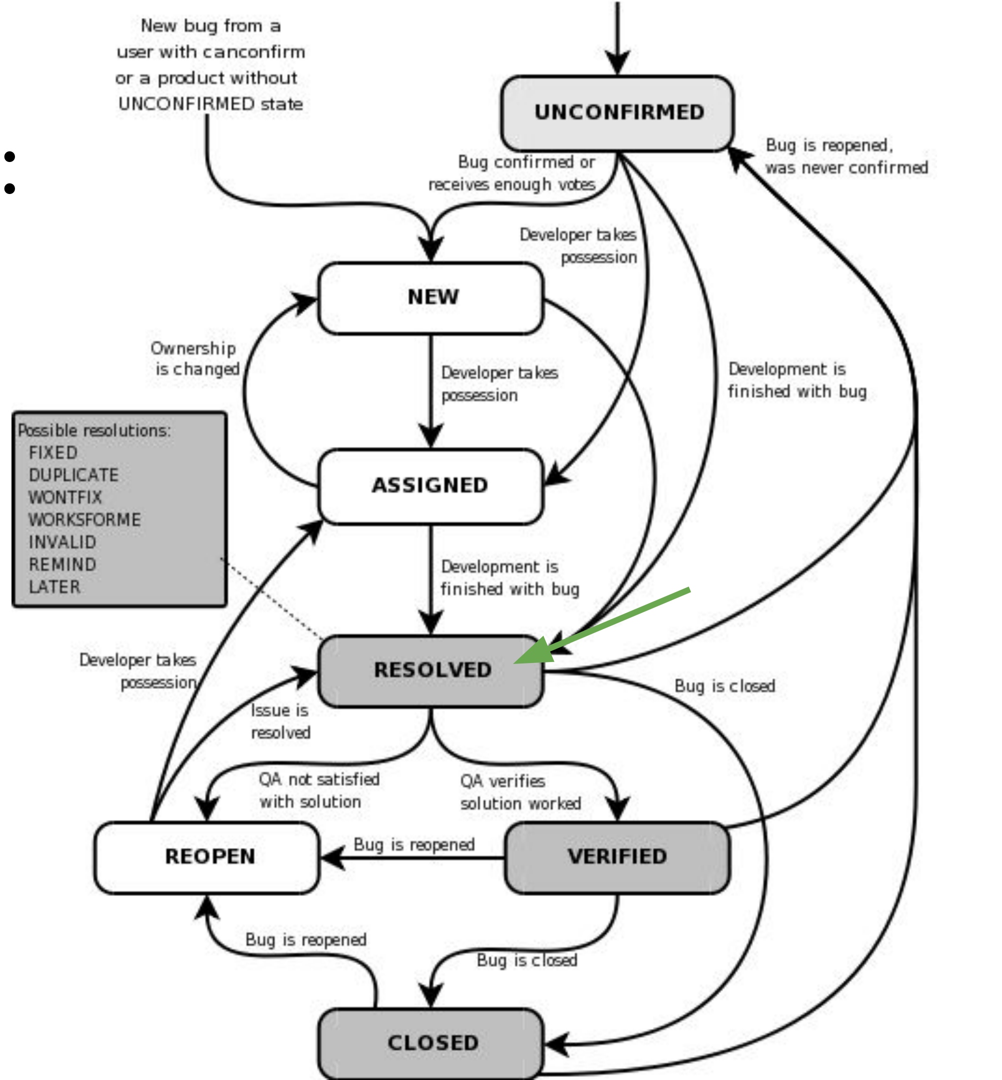
Defect report lifecycle: reopening

> This paper presents a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases including Linux, OpenSolaris, FreeBSD and also a mature *commercial* OS developed and evolved over the last *12* years, investigating not only the mistake patterns during bug-fixing but also the possible *human reasons* in the development process when these incorrect bug-fixes were introduced. Our major findings include: (1) at least 14.8%~24.4% of sampled fixes for post-release bugs [1] in these large OSes are incorrect and have made impacts to end users. (2) Among several common bug types, concurrency bugs are the most difficult to fix correctly: 39% of concurrency bug fixes are incorrect. (3) Developers and reviewers for incorrect fixes usually do not have enough knowledge about the involved code. For example, 27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix. Our results provide useful guidelines to design new tools and also to improve the development process. Based on our findings, the commercial software

- Many fixes are **wrong**, even on mature, critical software!
- Implication: reopening bugs is **common**
  - Importance of **regression testing**!

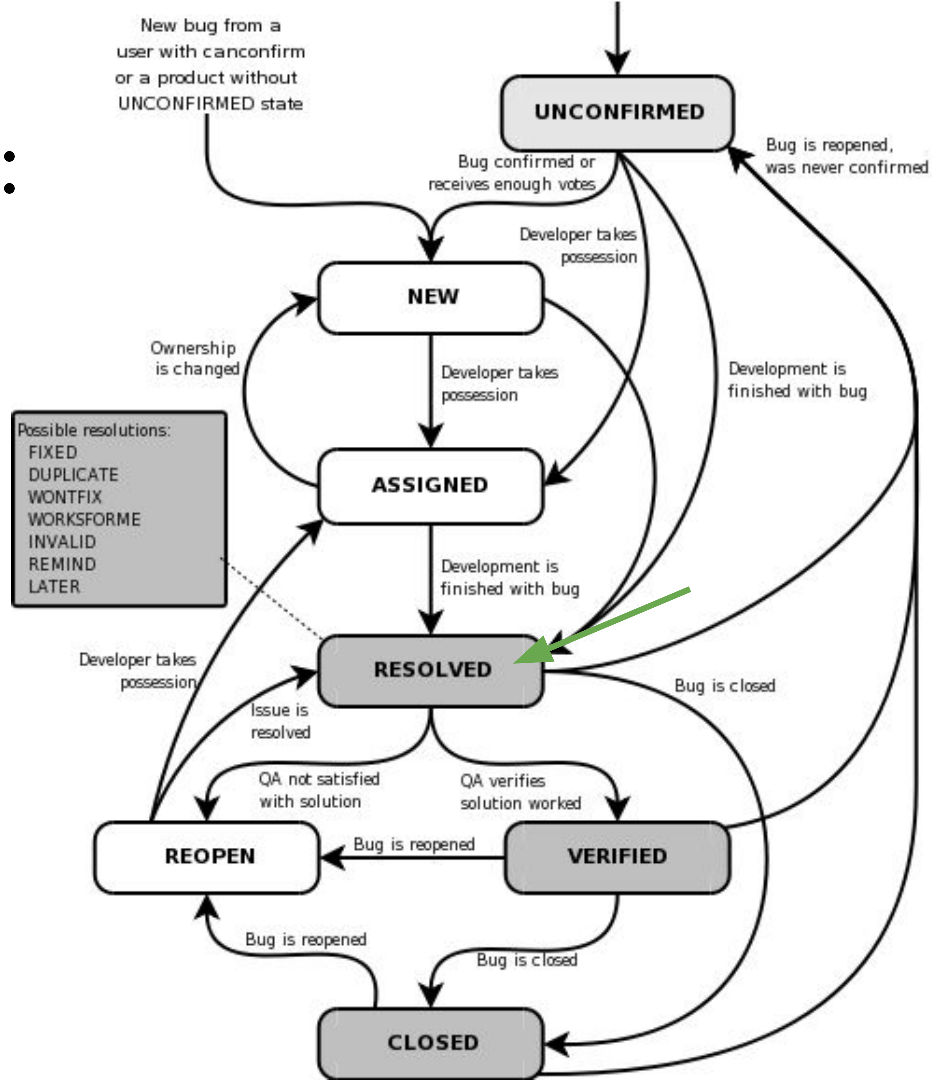
[Yin et al. How Do Fixes Become Bugs? ESEC/FSE 2011.]

Defect report lifecycle: fixing



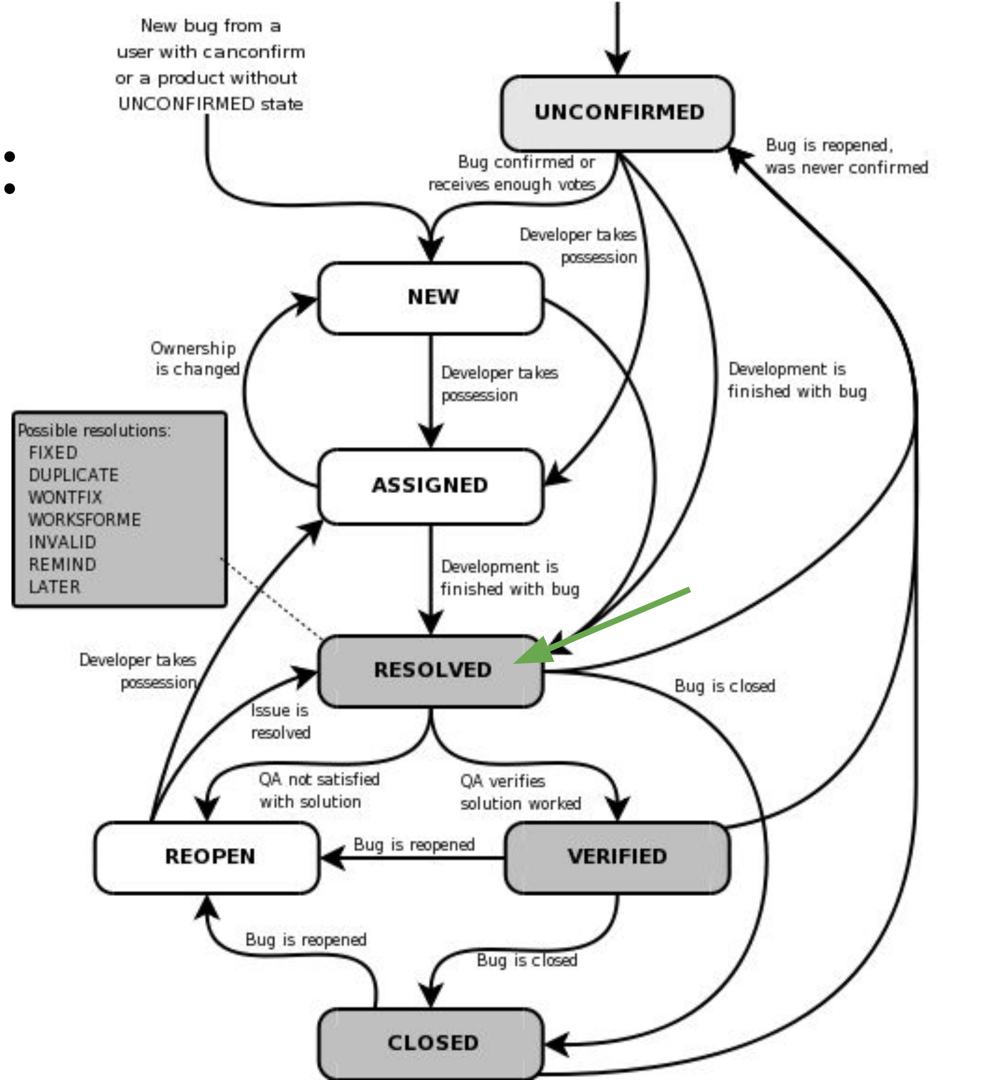
Defect report lifecycle: fixing

- Key question: once we have a good defect report, **how** do we figure out how to resolve the defect?



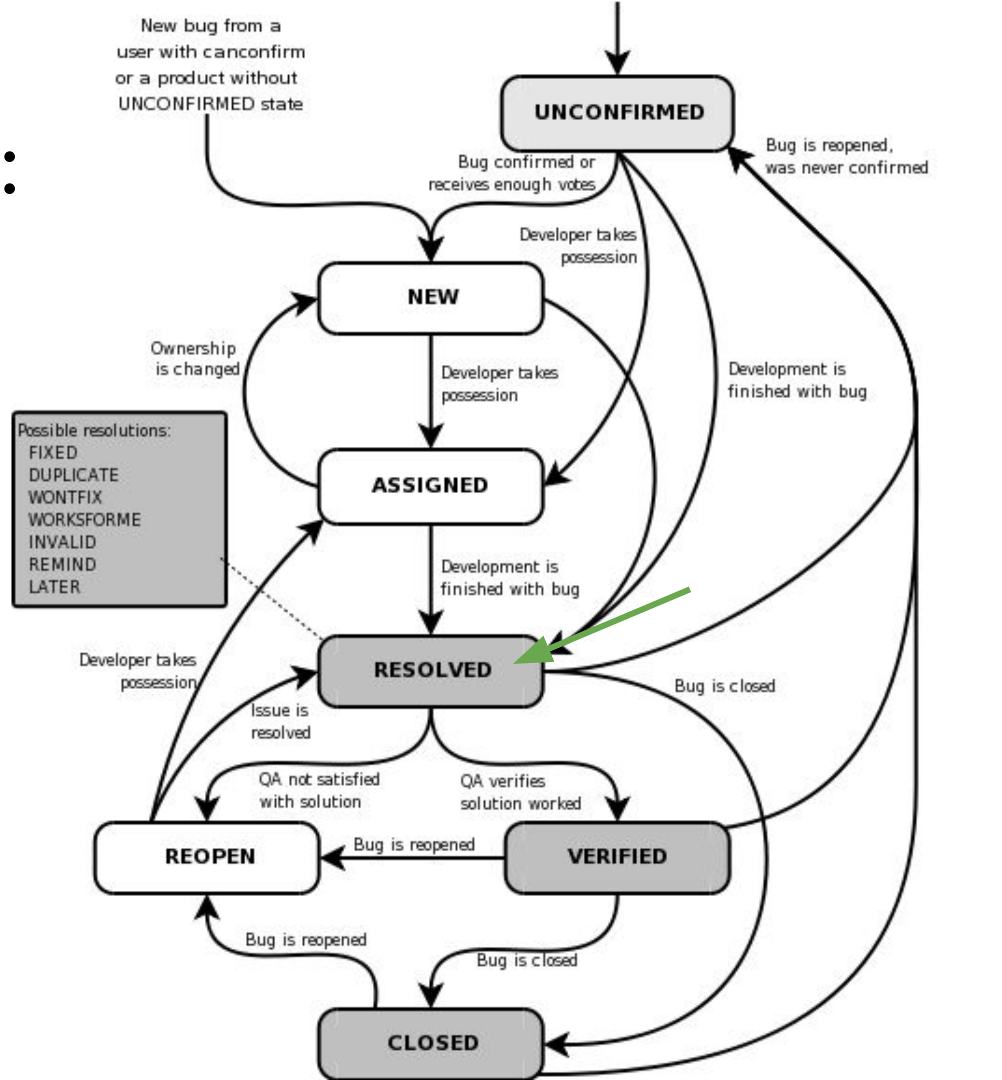
Defect report lifecycle: fixing

- Key question: once we have a good defect report, **how** do we figure out how to resolve the defect?
  - This is ***debugging***



Defect report lifecycle: fixing

- Key question: once we have a good defect report, **how** do we figure out how to resolve the defect?
    - This is ***debugging***
    - Rest of today's lecture + all of Friday's lecture on debugging



Debugging (Part 1/2)

Today’s agenda:

- What is a bug, anyway?
- Bug reports, triage, and the defect lifecycle
- **Debugging**
  - printf debugging and logging
  - delta debugging
  - debuggers

Debugging: what makes it difficult?

Debugging: what makes it difficult?

- modern software is **unimaginably huge**

Debugging: what makes it difficult?

- modern software is **unimaginably huge**
  - analogy: scale of space vs human scale
    - “Space is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space.” – Douglas Adams

Debugging: what makes it difficult?

- modern software is **unimaginably huge**
  - analogy: scale of space vs human scale
    - “Space is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space.” – Douglas Adams
  - you will be asked to fix bugs in very large software!

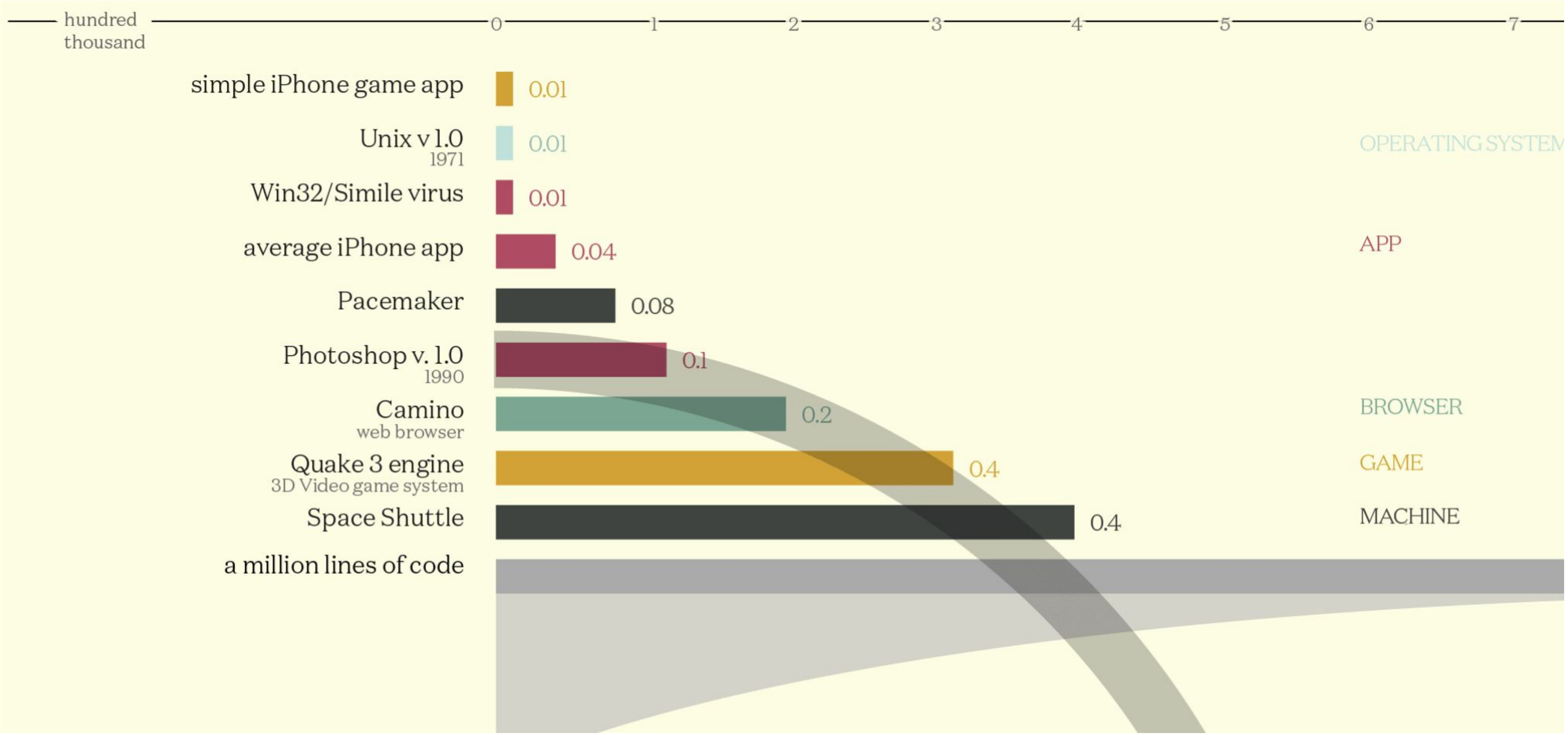
Debugging: what makes it difficult?

- modern software is **unimaginably huge**
  - analogy: scale of space vs human scale
    - “Space is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space.” – Douglas Adams
  - you will be asked to fix bugs in very large software!
- Techniques developed based on smaller code bases simply **do not apply** or scale to larger code bases

Debugging: what makes it difficult?

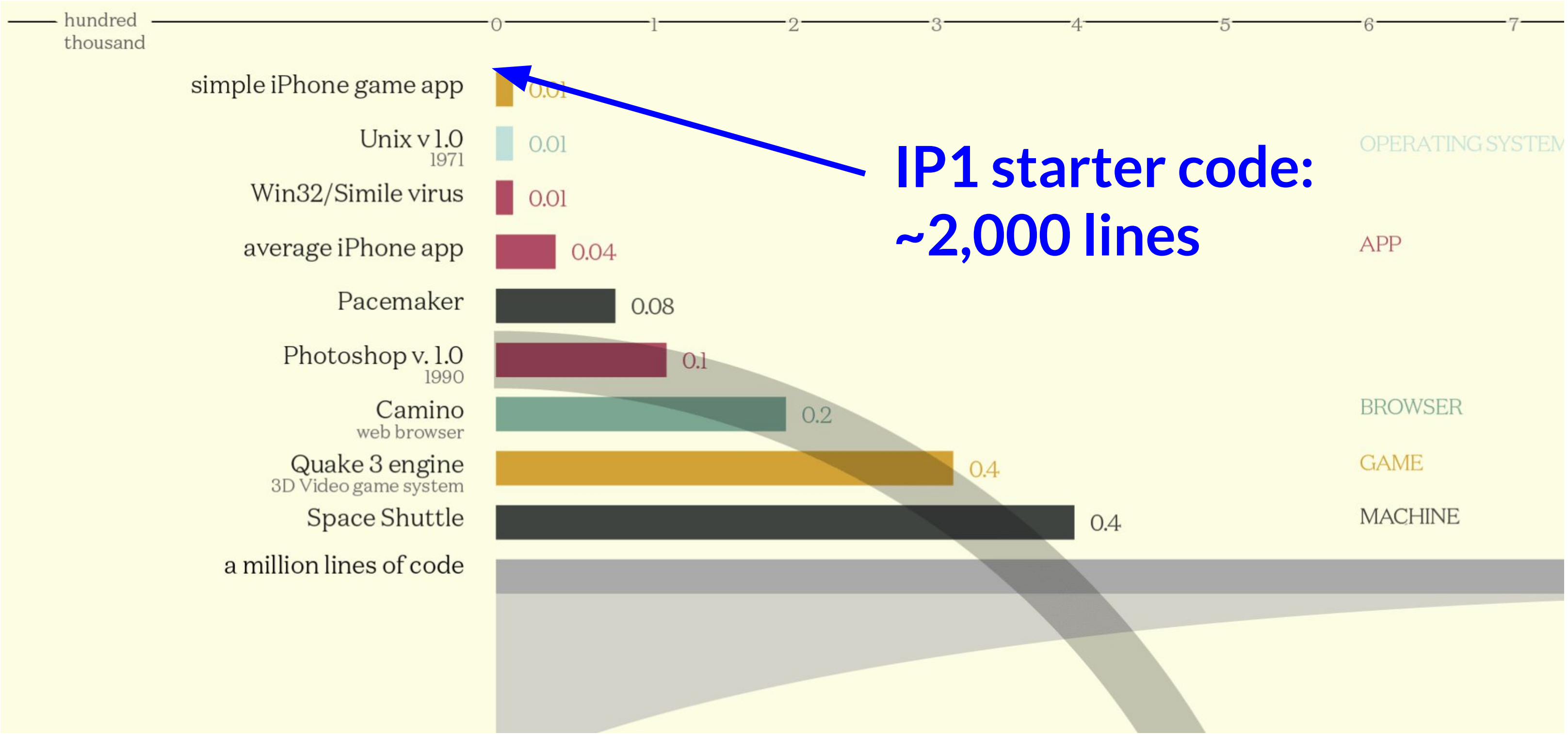
- modern software is **unimaginably huge**
  - analogy: scale of space vs human scale
    - “Space is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space.” – Douglas Adams
  - you will be asked to fix bugs in very large software!
- Techniques developed based on smaller code bases simply **do not apply** or scale to larger code bases
  - Techniques from the 1980s or your habits from classes

How big are programs, really?



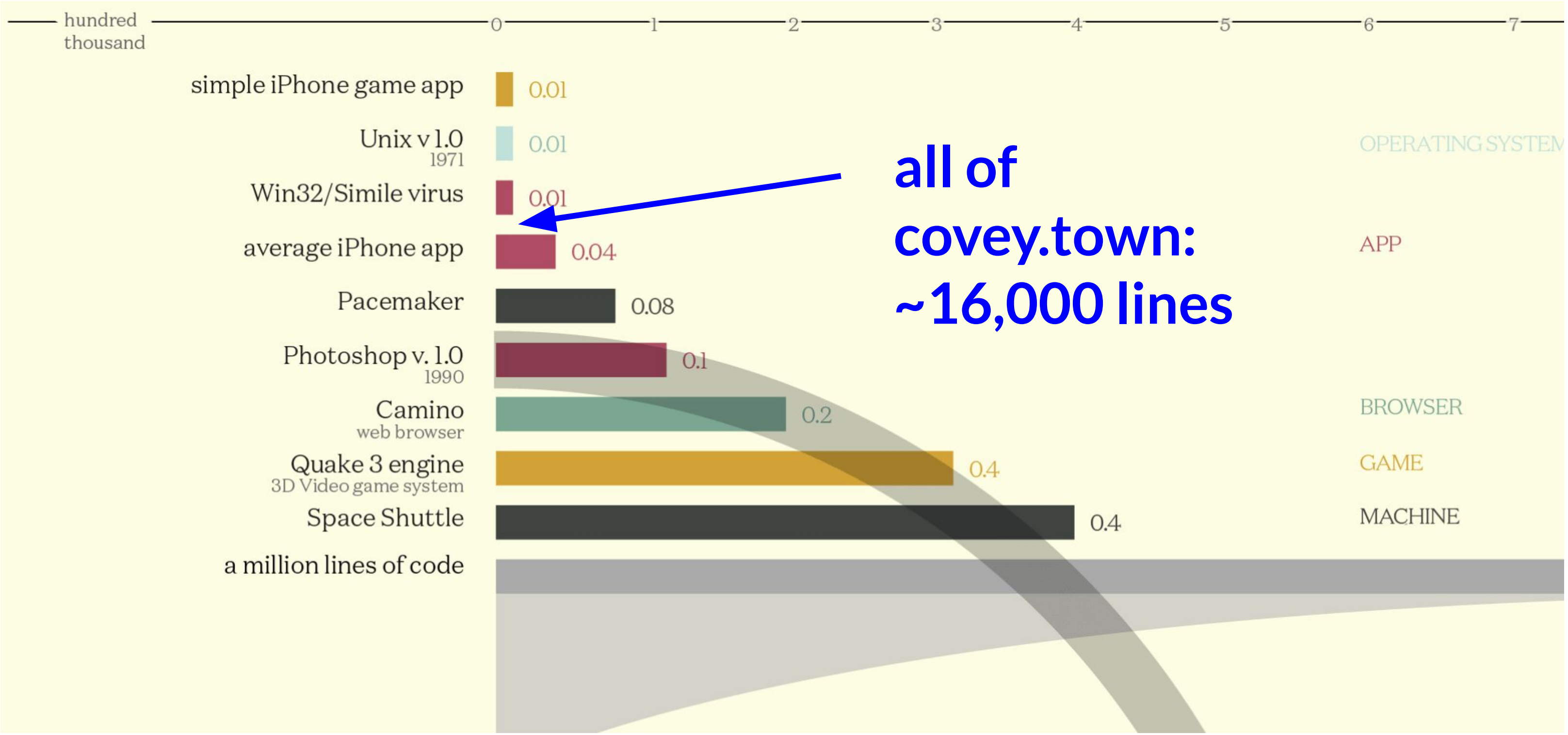
https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?



https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

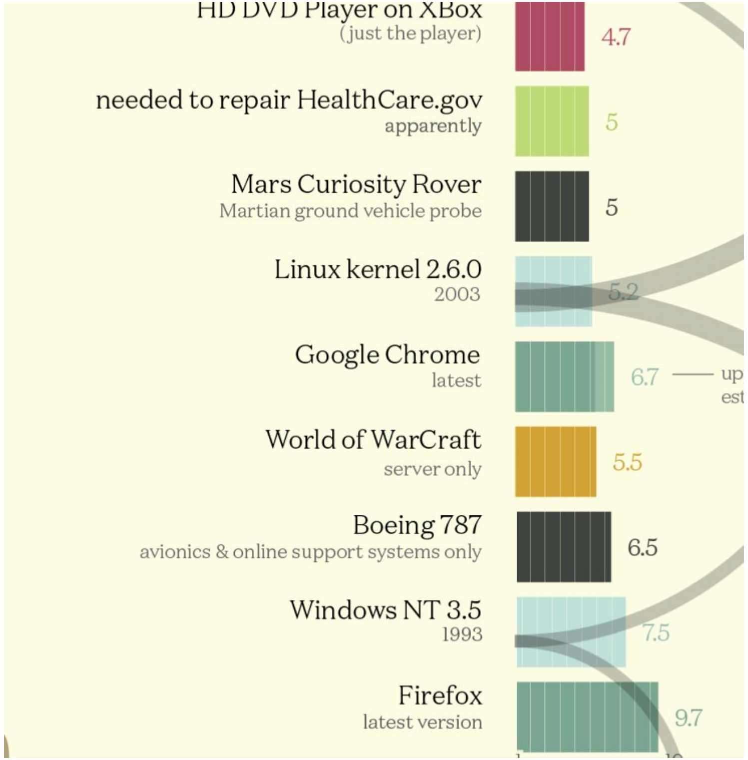
How big are programs, really?



all of covey.town: ~16,000 lines

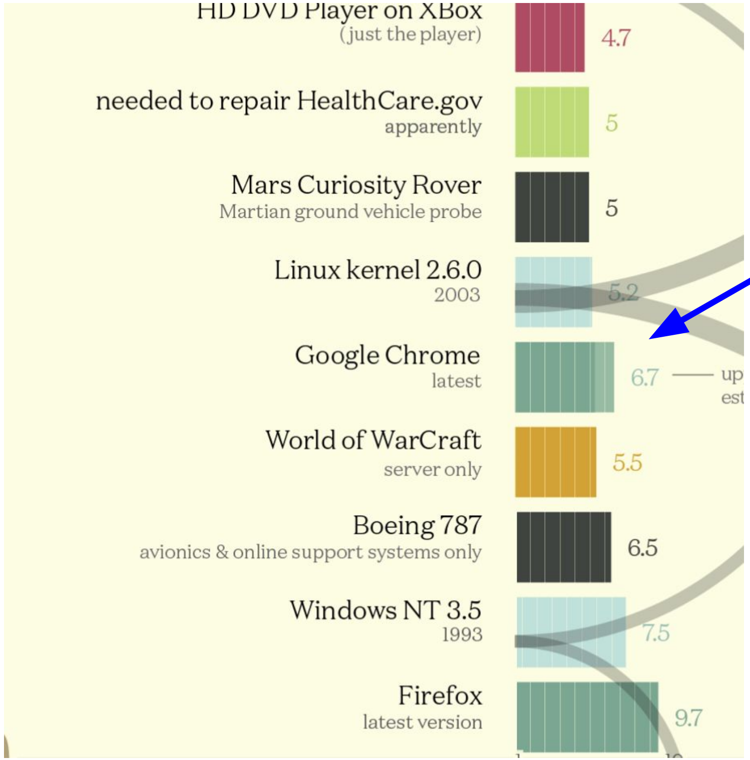
https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?



https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

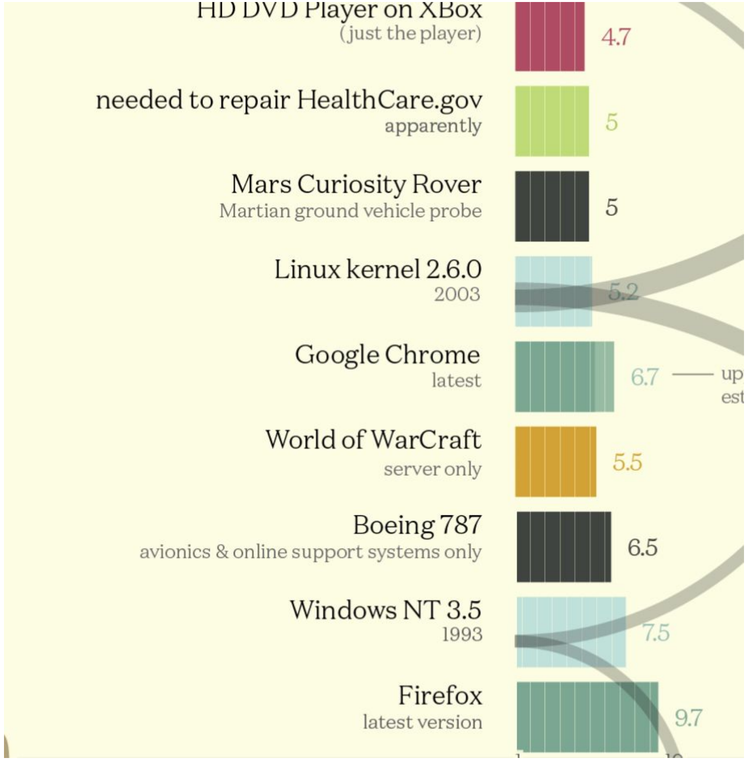
How big are programs, really?

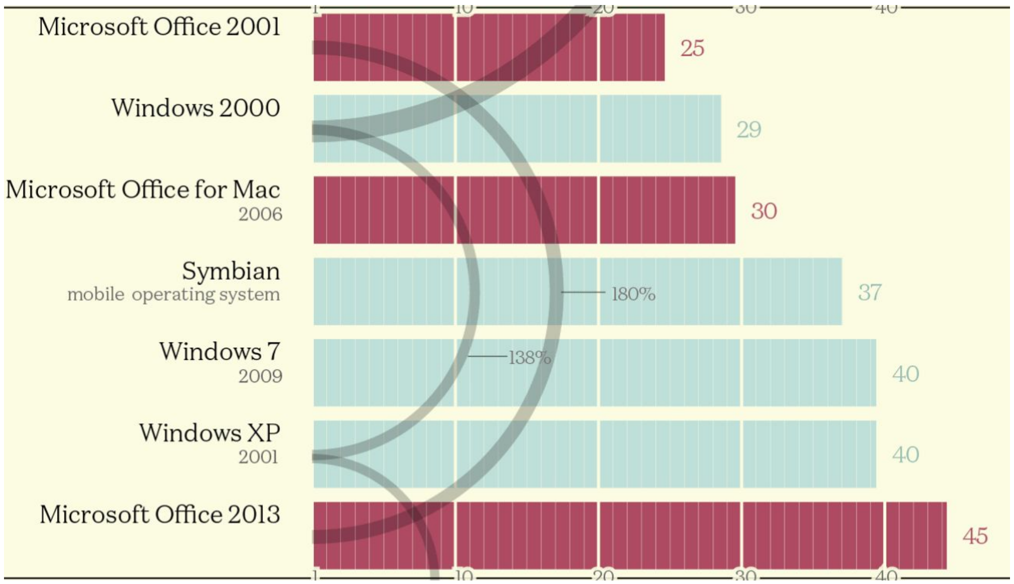


Chrome at ~7M LoC is ~400x bigger than covey.town

https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

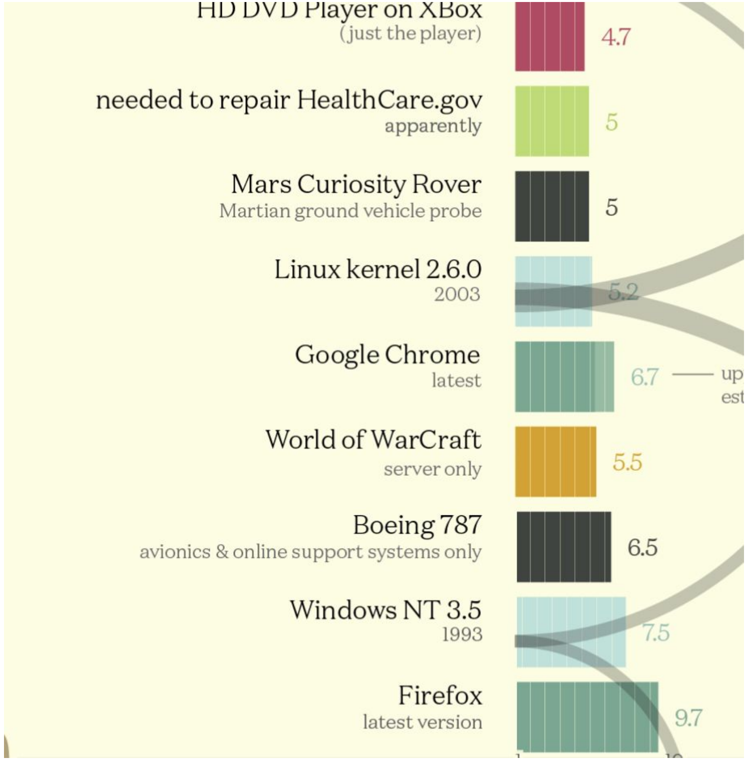
How big are programs, really?



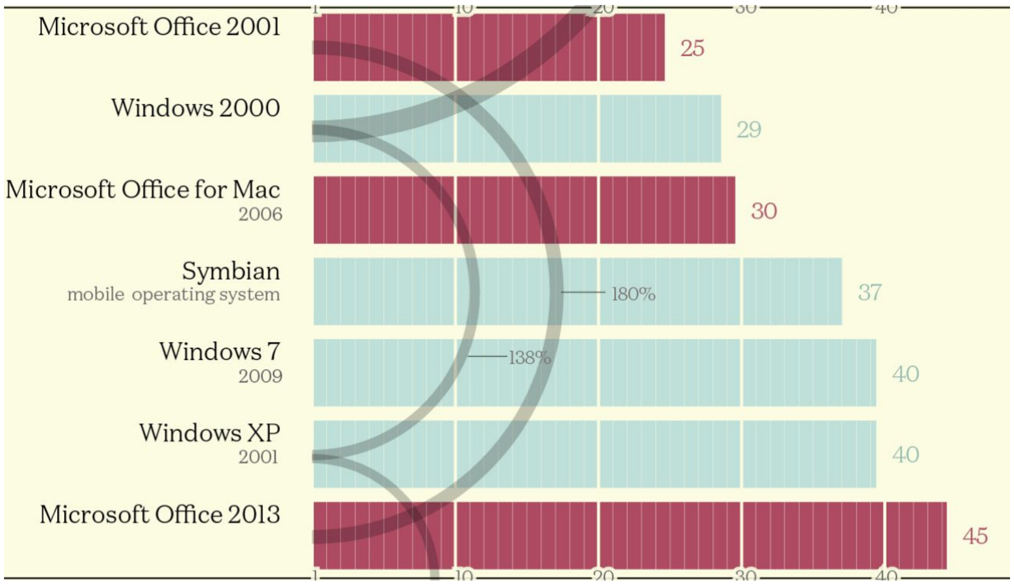


https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?

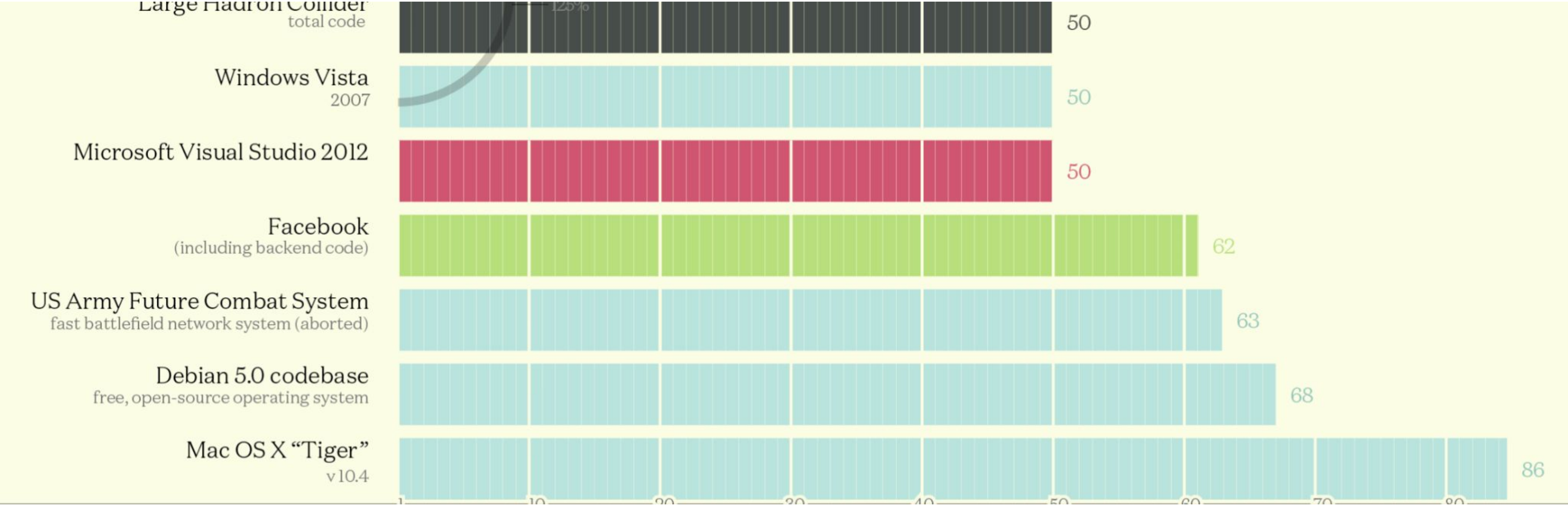


Chrome is small compared to even old versions of Windows!



https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?



https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?

WIRED

Google Is 2 Billion Lines of Code—And It's All in One Place

BUSINESS | CULTURE | GEAR | IDEAS | SCIENCE

SHARE

SHARE

TWEET

CADE METZ BUSINESS 09.16.15 10:00 AM

GOOGLE IS 2 BILLION LINES OF CODE—AND IT'S ALL IN ONE PLACE

https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/

Humans are poor at comprehending large scales

covey.town	16 000
google	2 000 000 000

Humans are poor at comprehending large scales

- covey.town 16 000
- google 2 000 000 000
- Imagine that there is a bug somewhere, **anywhere**, in covey.town

Humans are poor at comprehending large scales

- covey.town 16 000
- google 2 000 000 000
- Imagine that there is a bug somewhere, **anywhere**, in covey.town
  - Imagine further that you can find that bug in **one minute**

Humans are poor at comprehending large scales

- covey.town 16 000
- google 2 000 000 000
- Imagine that there is a bug somewhere, **anywhere**, in covey.town
  - Imagine further that you can find that bug in **one minute**
- At the same rate, it would take you **more than a month** to find it in all of google

Humans are poor at comprehending large scales

- covey.town 16 000
- google 2 000 000 000
- Imagine that there is a bug somewhere, **anywhere**, in covey.town
  - Imagine further that you can find that bug in **one minute**
- At the same rate, it would take you **more than a month** to find it in all of google
  - a one-hour bug on covey.town would take **years** on google!

Steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**

Steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**
- To effectively debug a problem, you should do the following:

Steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**
- To effectively debug a problem, you should do the following:
  - **reproduce** the issue yourself

Steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**
- To effectively debug a problem, you should do the following:
  - **reproduce** the issue yourself
  - **minimize** the reproduction so that you can reason about it

Steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**
- To effectively debug a problem, you should do the following:
    - **reproduce** the issue yourself
    - **minimize** the reproduction so that you can reason about it
    - **localize** the fault to a particular part of the program

Steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**
- To effectively debug a problem, you should do the following:
  - **reproduce** the issue yourself
  - **minimize** the reproduction so that you can reason about it
  - **localize** the fault to a particular part of the program
  - **test** possible fixes to find the right one

Steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**
- To effectively debug a problem, you should do the following:
    - **reproduce** the issue yourself
    - **minimize** the reproduction so that you can reason about it
    - **localize** the fault to a particular part of the program
    - **test** possible fixes to find the right one
    - **confirm** that your fix actually resolves the issue

Reproducing a bug

Definition: a bug can be ***reproduced*** if a developer can elicit the reported symptoms themself

Reproducing a bug

Definition: a bug can be ***reproduced*** if a developer can elicit the reported symptoms themself

- "**reported symptoms**" = "the problem described in the defect report"

Reproducing a bug

Definition: a bug can be ***reproduced*** if a developer can elicit the reported symptoms themself

- "**reported symptoms**" = "the problem described in the defect report"
- reproducing bugs is a **test input generation** problem:
  - find the inputs that cause the fault to occur

Reproducing a bug

Definition: a bug can be ***reproduced*** if a developer can elicit the reported symptoms themself

- "**reported symptoms**" = "the problem described in the defect report"
- reproducing bugs is a **test input generation** problem:
  - find the inputs that cause the fault to occur
- lots of bugs are resolved at this stage:

Reproducing a bug

Definition: a bug can be ***reproduced*** if a developer can elicit the reported symptoms themself

- "**reported symptoms**" = "the problem described in the defect report"
- reproducing bugs is a **test input generation** problem:
  - find the inputs that cause the fault to occur
- lots of bugs are resolved at this stage:
  - **WORKSFORME** is the BugZilla resolution for this

Reproducing a bug

Definition: a bug can be ***reproduced*** if a developer can elicit the reported symptoms themself

- "**reported symptoms**" = "the problem described in the defect report"
- reproducing bugs is a **test input generation** problem:
    - find the inputs that cause the fault to occur
- lots of bugs are resolved at this stage:
    - **WORKSFORME** is the BugZilla resolution for this
    - especially bugs reported by users often do not get past this stage: **not enough information** to reproduce the fault

Minimizing the reproduction

Definition: a ***minimal reproduction*** of a bug is the smallest input that elicits the bug's reported symptoms

Minimizing the reproduction

Definition: a ***minimal reproduction*** of a bug is the smallest input that elicits the bug's reported symptoms

- defect reports containing minimal failing examples are the **gold standard** (but rare in practice)

Minimizing the reproduction

Definition: a ***minimal reproduction*** of a bug is the smallest input that elicits the bug’s reported symptoms

- defect reports containing minimal failing examples are the **gold standard** (but rare in practice)
- commonly, even reproducible bugs come with a **complex test input**
    - e.g., including the entire environment in which the software was running

Minimizing the reproduction

Definition: a ***minimal reproduction*** of a bug is the smallest input that elicits the bug's reported symptoms

- defect reports containing minimal failing examples are the **gold standard** (but rare in practice)
- commonly, even reproducible bugs come with a **complex test input**
  - e.g., including the entire environment in which the software was running
- minimizing the reproduction helps the developer reason about **which part** of the software might be responsible for the bug

Minimizing the reproduction

Definition: a ***minimal reproduction*** of a bug is the smallest input that elicits the bug's reported symptoms

- defect reports containing minimal failing examples are the **gold standard** (but rare in practice)
- commonly, even reproducible bugs come with a **complex test input**
  - e.g., including the entire environment in which the software was running
- minimizing the reproduction helps the developer reason about **which part** of the software might be responsible for the bug
  - also useful for **assignment**

Minimizing the reproduction

Definition: a ***minimal reproduction*** of a

elicits the bug's reported symptoms

- defect reports containing minima
  standard (but rare in practice)
- commonly, even reproducible bugs come with a **complex test input**
  - e.g., including the entire environment in which the software was running
- minimizing the reproduction helps the developer reason about **which part** of the software might be responsible for the bug
  - also useful for **assignment**

Minimizing the reproduction is **sometimes unnecessary**: a small (but not minimal) input is often good enough

Fault localization

Definition: ***fault localization*** is the task of identifying source code regions implicated in a bug

Fault localization

Definition: ***fault localization*** is the task of identifying source code regions implicated in a bug

- “This regression test is failing. Which lines should we change to fix things?”

Fault localization

Definition: ***fault localization*** is the task of identifying source code regions implicated in a bug

- "This regression test is failing. Which lines should we change to fix things?"
- Answer is **not unique**: there are often many places to fix a bug

Fault localization

Definition: ***fault localization*** is the task of identifying source code regions implicated in a bug

- "This regression test is failing. Which lines should we change to fix things?"
- Answer is **not unique**: there are often many places to fix a bug
  - Example: check for null at caller or callee?

Fault localization

Definition: ***fault localization*** is the task of identifying source code regions implicated in a bug

- "This regression test is failing. Which lines should we change to fix things?"
- Answer is **not unique**: there are often many places to fix a bug
  - Example: check for null at caller or callee?
- While some tool support is available, state of the practice is **manual**

Fault localization

Definition: ***fault localization*** is the task of identifying source code regions implicated in a bug

- "This regression test is failing. Which lines should we change to fix things?"
- Answer is **not unique**: there are often many places to fix a bug
  - Example: check for null at caller or callee?
- While some tool support is available, state of the practice is **manual**
  - automated tools rank parts of the program by "**suspiciousness**"

Fault localization

Definition: ***fault localization*** is the task of identifying source code regions implicated in a bug

- "This regression test is failing. Which lines should we change to fix things?"
- Answer is **not unique**: there are often many places to fix a bug
  - Example: check for null at caller or callee?
- While some tool support is available, state of the practice is **manual**
  - automated tools rank parts of the program by "**suspiciousness**"
  - suspiciousness computed by how often each part of the program is **covered** by passing vs. failing tests

Testing and confirming your fix

Testing and confirming your fix

- rule of thumb: every bug fix should be accompanied by a **new regression test**

Testing and confirming your fix

- rule of thumb: every bug fix should be accompanied by a **new regression test**
  - often **more than one**: many fixes are possible, but some are better than others, so you want tests that rule out “wrong” fixes that you tried

Testing and confirming your fix

- rule of thumb: every bug fix should be accompanied by a **new regression test**
  - often **more than one**: many fixes are possible, but some are better than others, so you want tests that rule out “wrong” fixes that you tried
- another rule of thumb: each new regression test should **fail before** applying your fix (and pass after, of course)

Testing and confirming your fix

- rule of thumb: every bug fix should be accompanied by a **new regression test**
  - often **more than one**: many fixes are possible, but some are better than others, so you want tests that rule out "wrong" fixes that you tried
- another rule of thumb: each new regression test should **fail before** applying your fix (and pass after, of course)
  - easy mistake to make: write or modify a test in such a way that you end up **no longer reproducing** the bug while "fixing" the bug

Testing and confirming your fix

- rule of thumb: every bug fix should be accompanied by a **new regression test**
  - often **more than one**: many fixes are possible, but some are better than others, so you want tests that rule out "wrong" fixes that you tried
- another rule of thumb: each new regression test should **fail before** applying your fix (and pass after, of course)
  - easy mistake to make: write or modify a test in such a way that you end up **no longer reproducing** the bug while "fixing" the bug
  - best practice: commit tests separately

Debugging (Part 2/2)

Two-lecture agenda:

- What is a bug, anyway?
- Bug reports, triage, and the defect lifecycle
- **Debugging**
    - printf debugging and logging
    - delta debugging
    - debuggers

Review: steps of debugging

- When working with very large systems, it is important to think of debugging **systematically**
- To effectively debug a problem, you should do the following:
    - **reproduce** the issue yourself
    - **minimize** the reproduction so that you can reason about it
    - **localize** the fault to a particular part of the program
    - **test** possible fixes to find the right one
    - **confirm** that your fix actually resolves the issue

Debugging strategies

- the remainder of our lectures on debugging will be devoted to discussing different **strategies** for debugging

Debugging strategies

- the remainder of our lectures on debugging will be devoted to discussing different **strategies** for debugging
- all of these strategies have one **key idea** in common: treat debugging as a series of **hypothesis tests**

Debugging strategies

- the remainder of our lectures on debugging will be devoted to discussing different **strategies** for debugging
- all of these strategies have one **key idea** in common: treat debugging as a series of **hypothesis tests**
    - hypothesis testing is one of the key components of the **scientific method**:

Debugging strategies

- the remainder of our lectures on debugging will be devoted to discussing different **strategies** for debugging
- all of these strategies have one **key idea** in common: treat debugging as a series of **hypothesis tests**
    - hypothesis testing is one of the key components of the **scientific method**:
        1. guess why something happens, devise an experiment to test if your guess is correct, then run the experiment
        2. repeat step 1 until you’ve figured it out

Debugging as hypothesis testing

- the key to treating debugging as hypothesis testing is to make **falsifiable guesses** about why the program is behaving a particular way

Debugging as hypothesis testing

- the key to treating debugging as hypothesis testing is to make **falsifiable guesses** about why the program is behaving a particular way
    - "***falsifiable***" = "can be true or false"

Debugging as hypothesis testing

- the key to treating debugging as hypothesis testing is to make **falsifiable guesses** about why the program is behaving a particular way
  - "***falsifiable***" = "can be true or false"
  - ideally, you'd also like your guesses to be easy to test

Debugging as hypothesis testing

- the key to treating debugging as hypothesis testing is to make **falsifiable guesses** about why the program is behaving a particular way
  - "***falsifiable***" = "can be true or false"
  - ideally, you'd also like your guesses to be easy to test
- each time you make such a guess, you need to **design an experiment** to check if the guess is correct

Debugging as hypothesis testing

- the key to treating debugging as hypothesis testing is to make **falsifiable guesses** about why the program is behaving a particular way
  - "***falsifiable***" = "can be true or false"
  - ideally, you'd also like your guesses to be easy to test
- each time you make such a guess, you need to **design an experiment** to check if the guess is correct
  - most of the debugging strategies we'll talk about are ways to check if a particular guess is correct

Debugging as hypothesis testing

- the key to treating debugging as hypothesis testing is to make **falsifiable guesses** about why the program is behaving a particular way
  - "***falsifiable***" = "can be true or
  - ideally, you'd also like your g
- each time you make such a gues **experiment** to check if the guess
  - most of the debugging strategies we'll talk about are ways to check if a particular guess is correct

Big difference between you ("**computer scientist**") and anyone who knows how to program: the ability to apply the **scientific method** to coding