Debugging (1/2)

Martin Kellogg

Debugging (Part 1/2)

Today's agenda:

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

Debugging (Part 1/2)

Today's agenda:

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

Reading quiz (debugging part 1)

Q1: **TRUE** or **FALSE**: a hypothesis is a guess that can be tested to see if it is true or not.

Q2: According to the author, the relationship between debugging and science is:

A. good programmers are usually good scientists, too
B. debuggers (e.g., gdb) implement the scientific method internally
C. both sometimes involve rubber ducks
D. each step of debugging is a miniature science experiment

Reading quiz (debugging part 1)

Q1: **TRUE** or **FALSE**: a hypothesis is a guess that can be tested to see if it is true or not.

Q2: According to the author, the relationship between debugging and science is:

A. good programmers are usually good scientists, too
B. debuggers (e.g., gdb) implement the scientific method internally
C. both sometimes involve rubber ducks
D. each step of debugging is a miniature science experiment

Reading quiz (debugging part 1)

Q1: **TRUE** or **FALSE**: a hypothesis is a guess that can be tested to see if it is true or not.

Q2: According to the author, the relationship between debugging and science is:

A. good programmers are usually good scientists, too
B. debuggers (e.g., gdb) implement the scientific method internally
C. both sometimes involve rubber ducks
D. each step of debugging is a miniature science experiment

Debugging (Part 1/2)

Today's agenda:

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

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”)

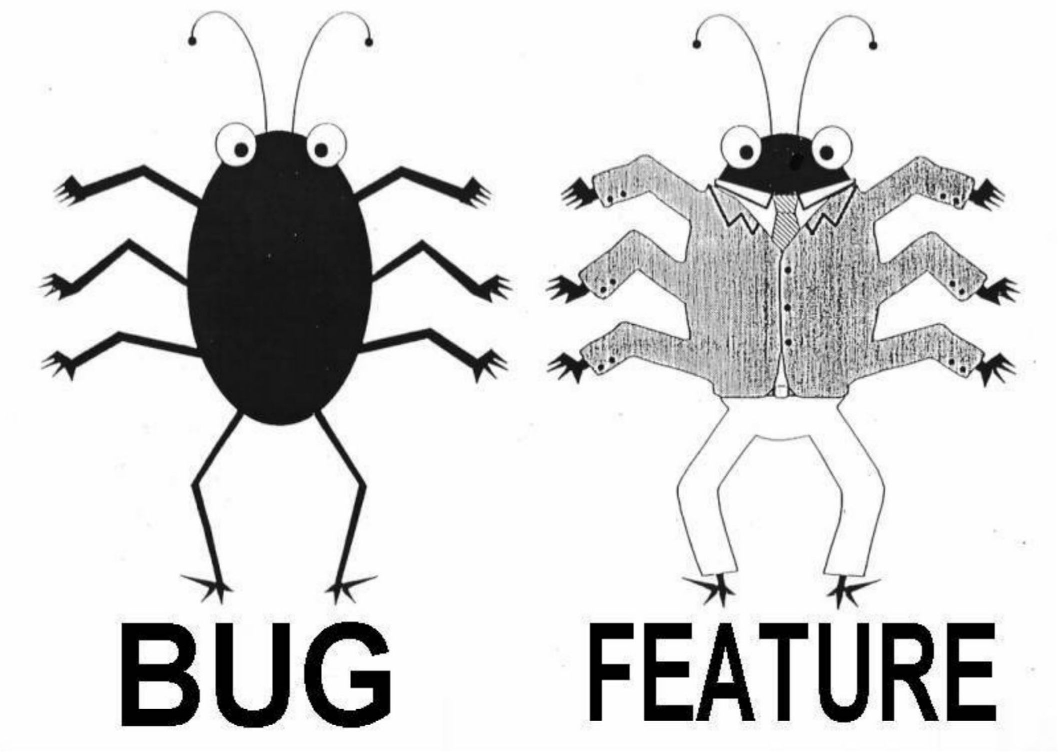
Terminology: bug vs. features

Terminology: bug vs. features

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

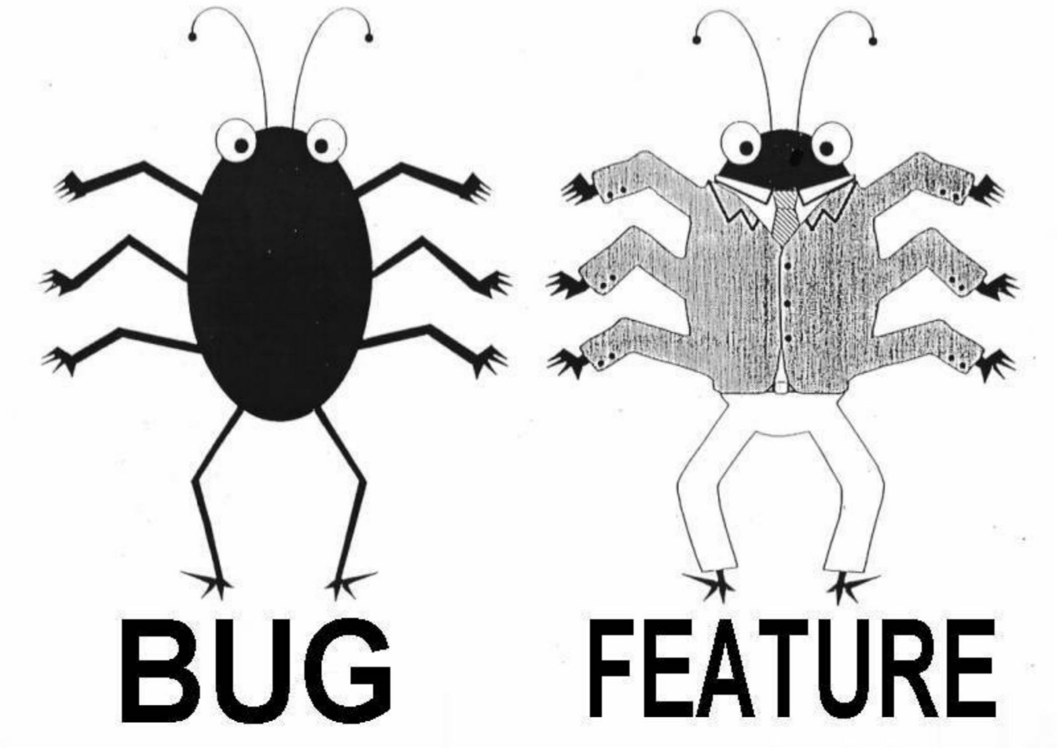
Terminology: bug vs. features

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



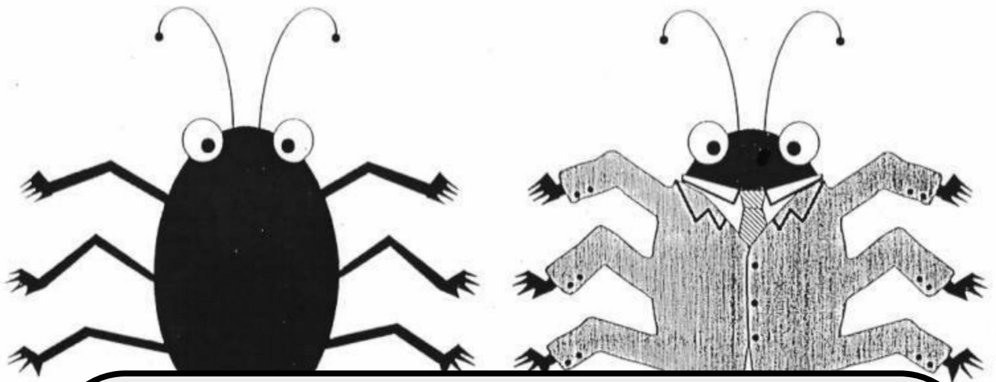
Terminology: bug vs. features

- 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:

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

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.

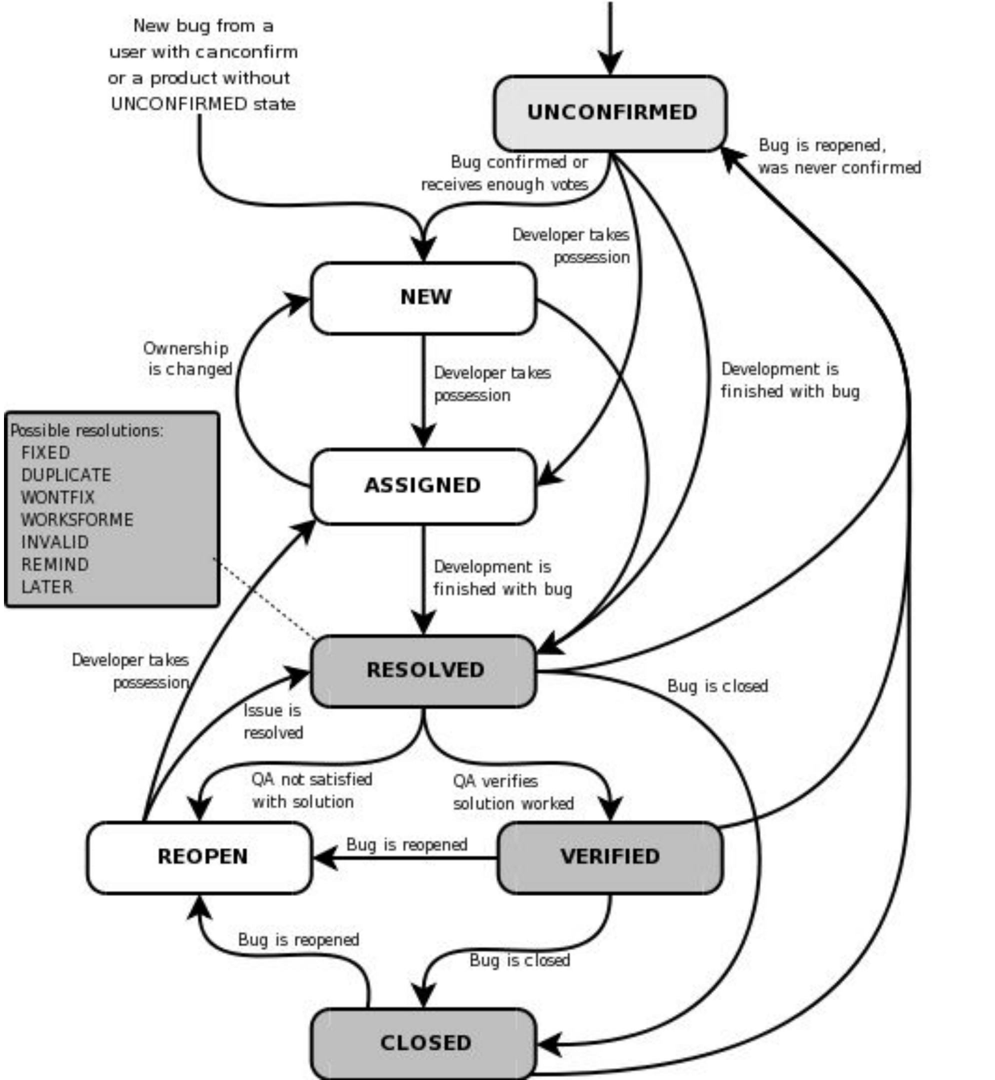
- Not every defect report follows the same path
- The overall process is **not linear**
  - There are multiple entry points, some cycles, and multiple exit points (and some never leave …)

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

Defect report lifecycle

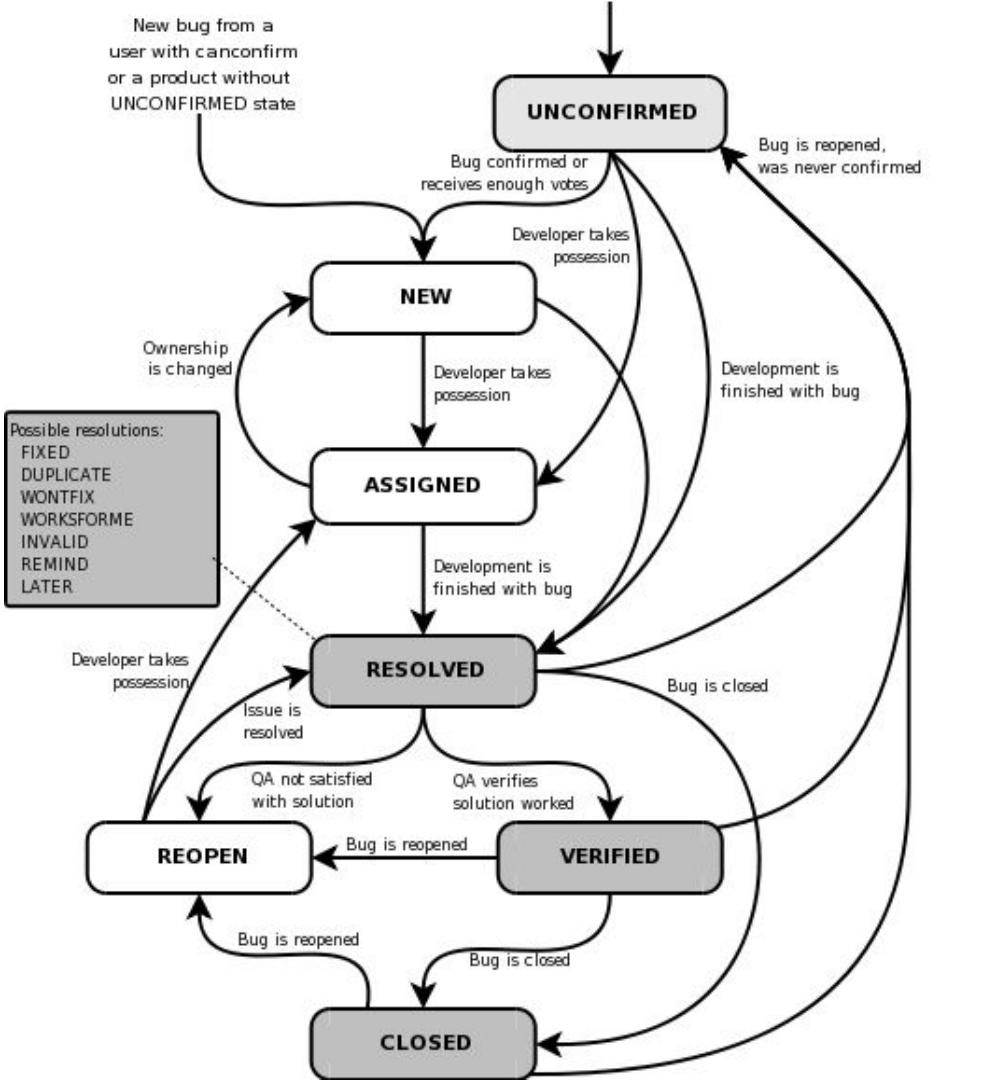
Defect report lifecycle

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



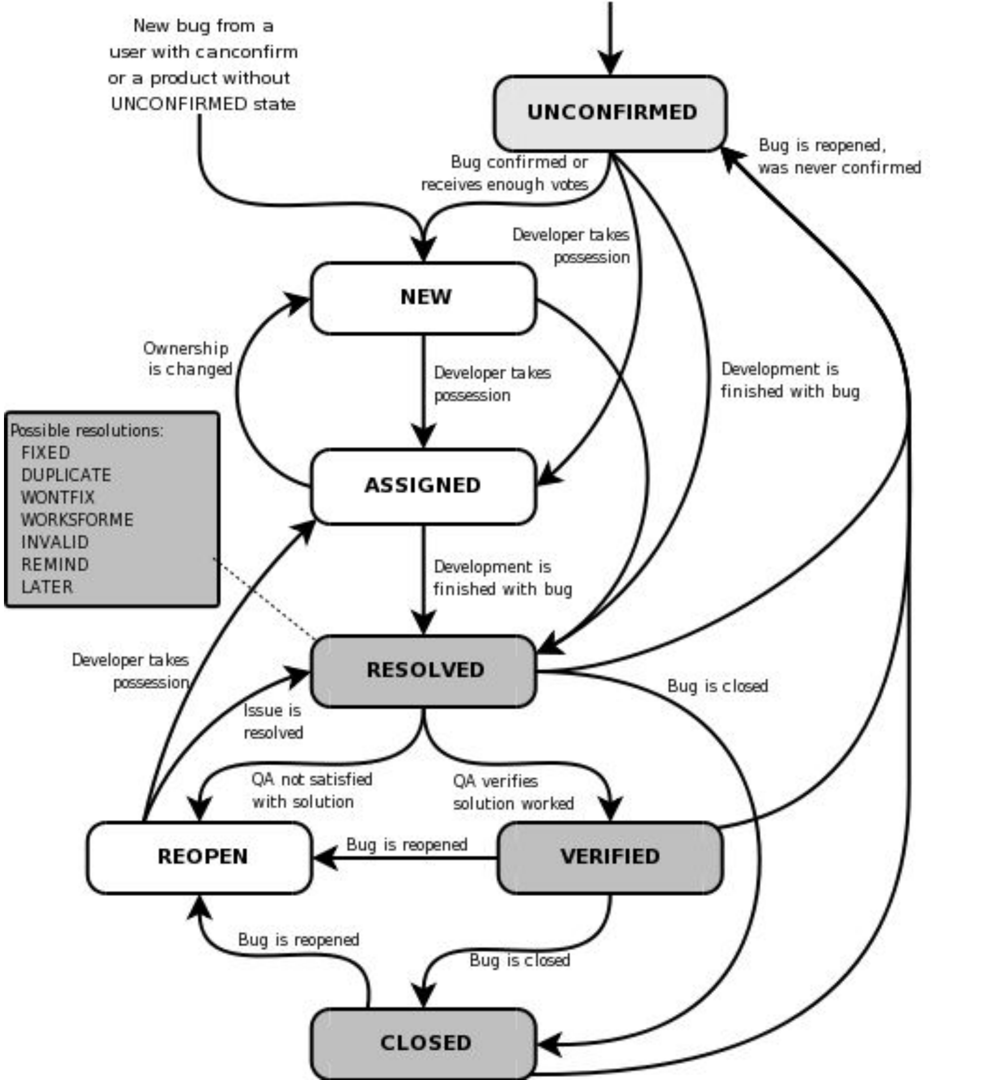
Defect report lifecycle

- 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)



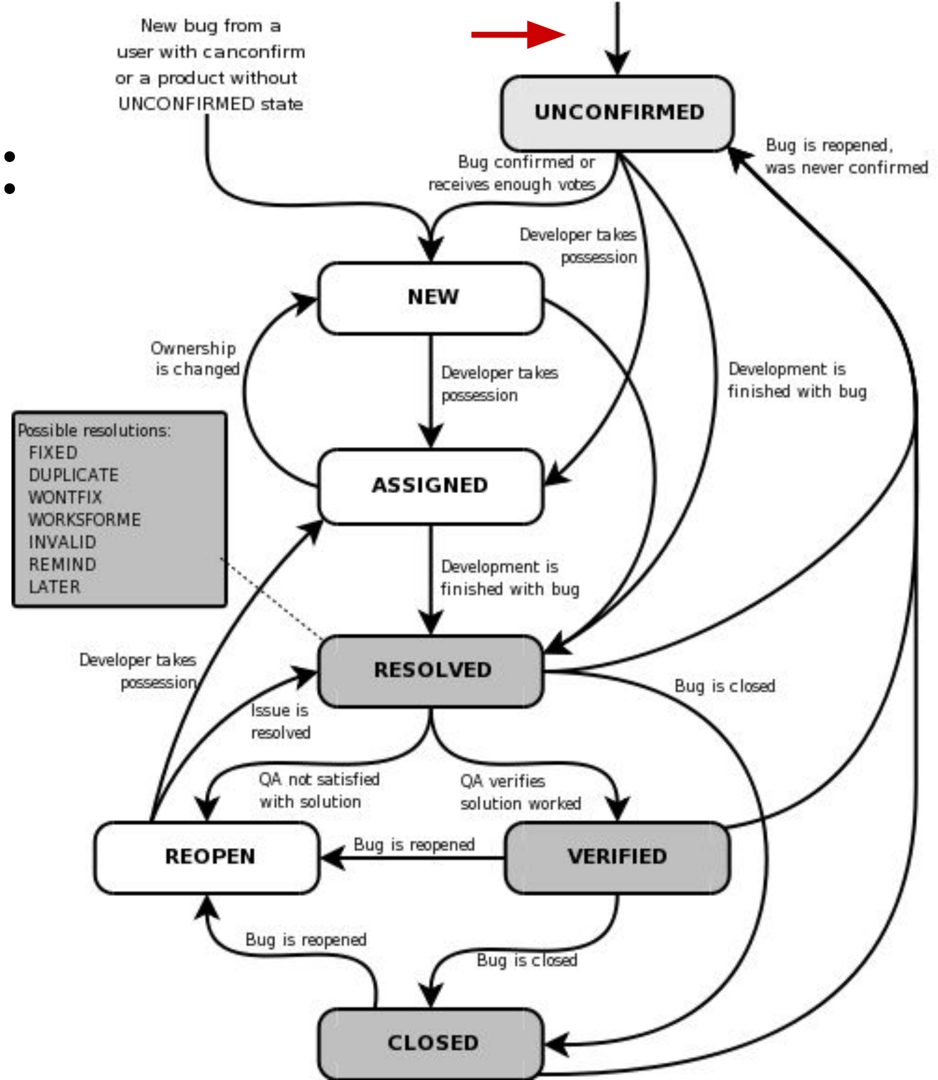
Defect report lifecycle

- 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)



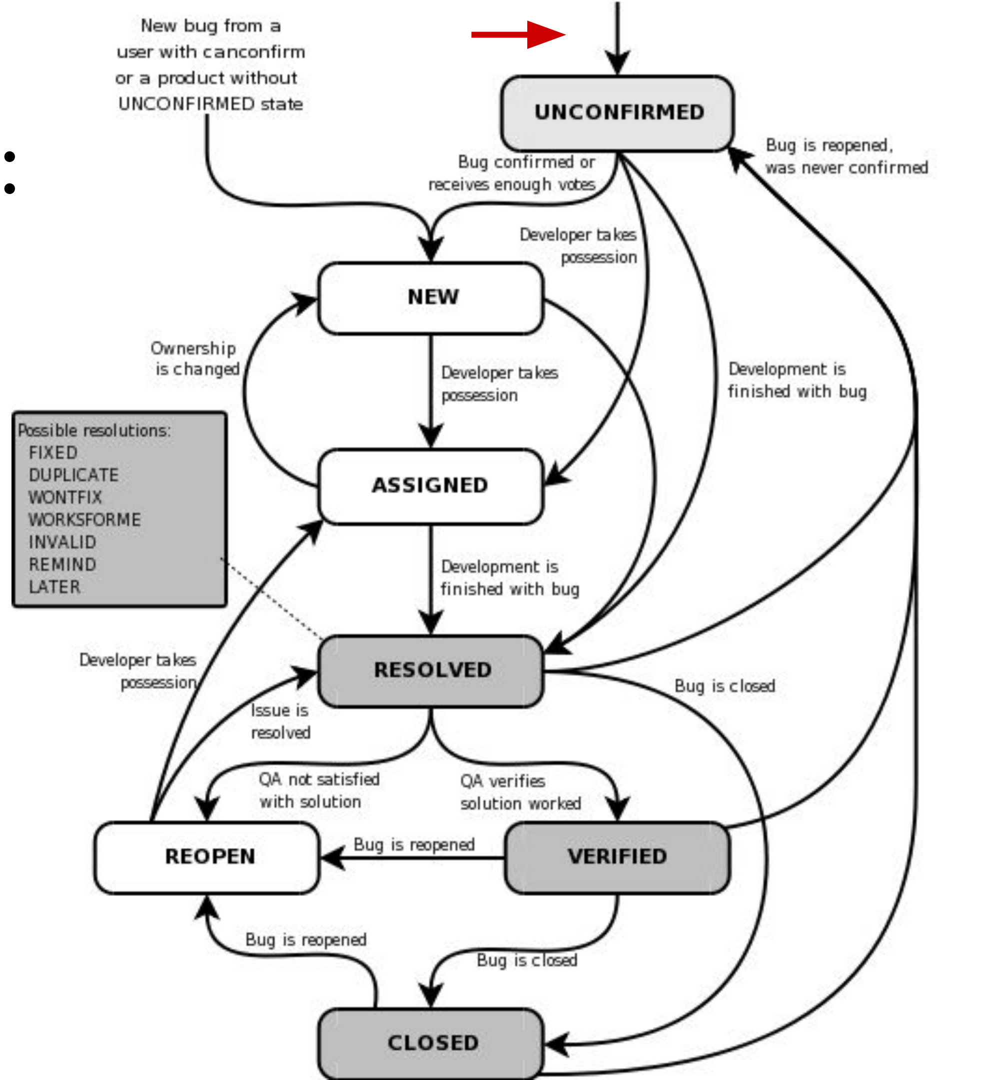
Defect report lifecycle: new bugs

- most new bugs enter the system as “unconfirmed”



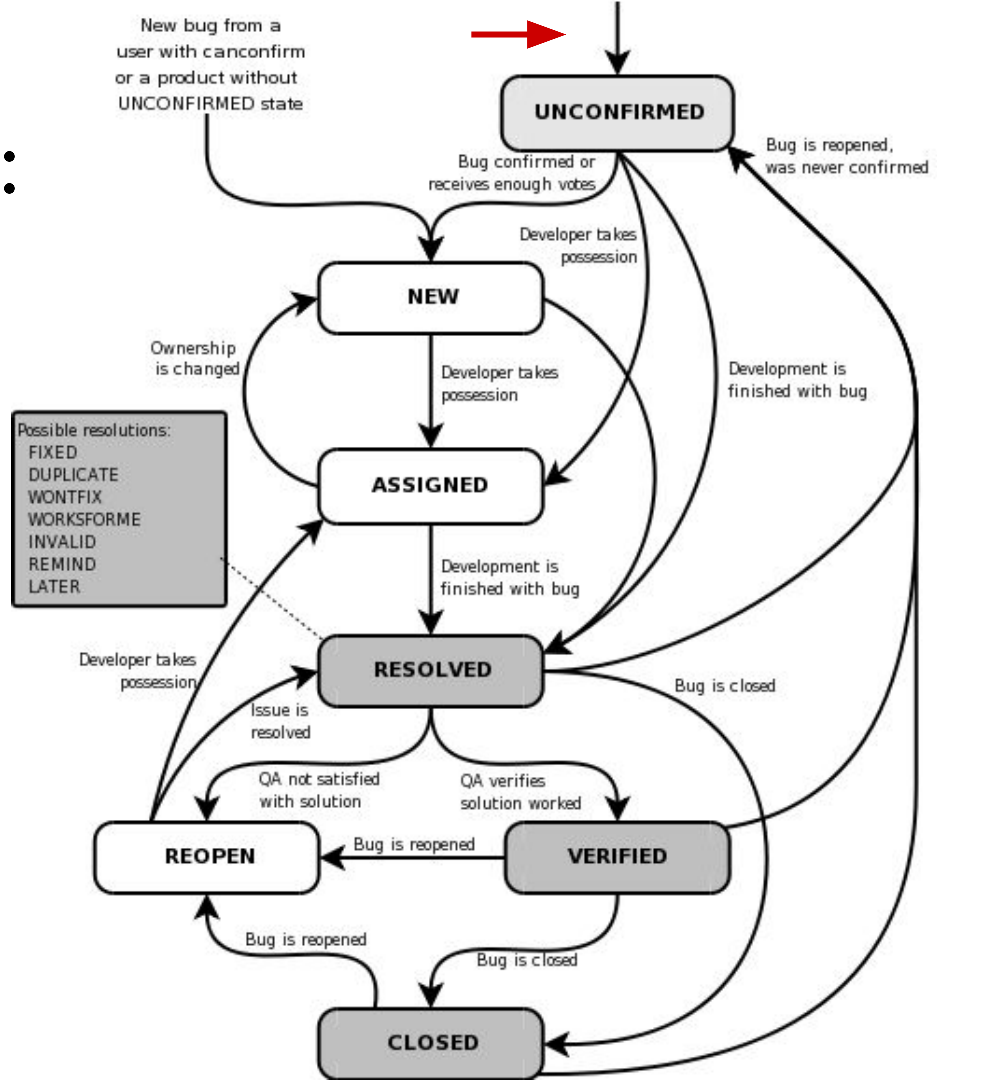
Defect report lifecycle: new bugs

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



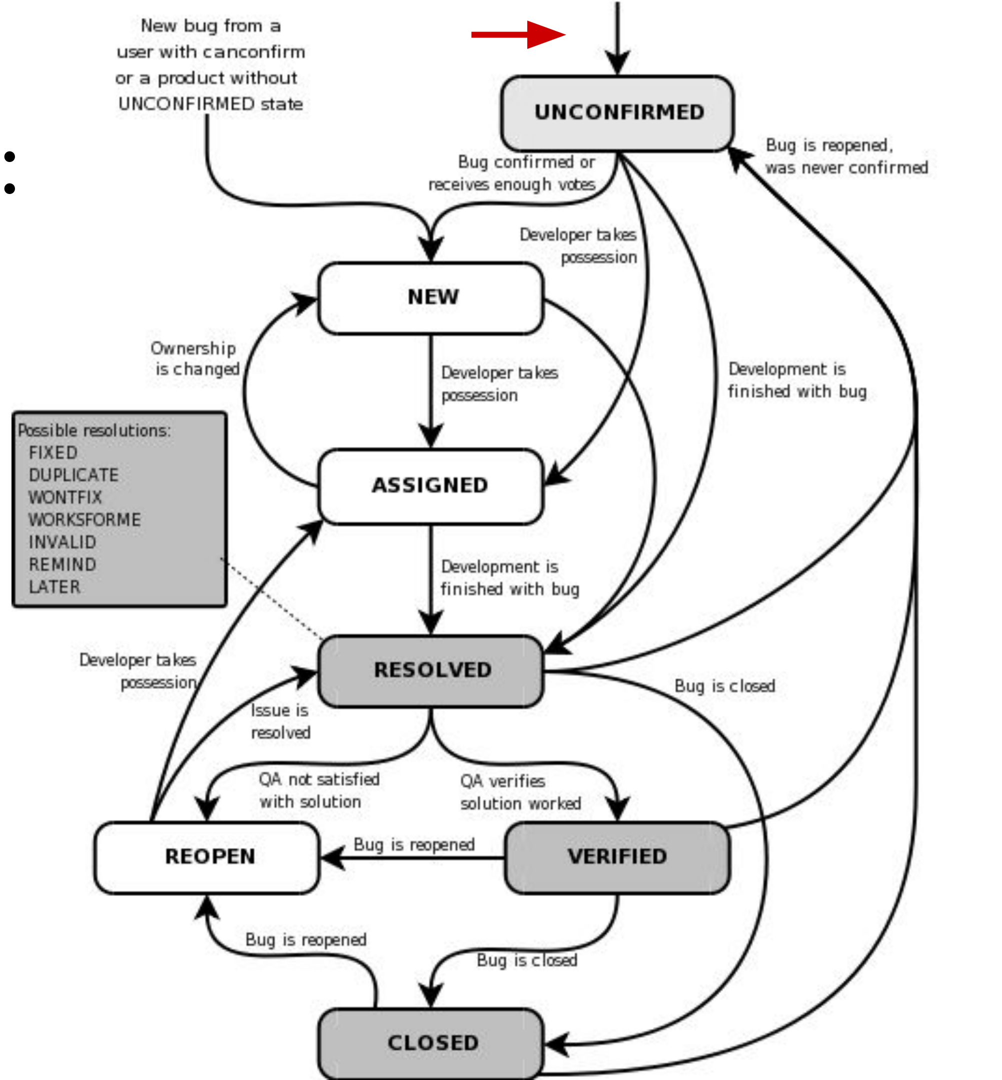
Defect report lifecycle: new bugs

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



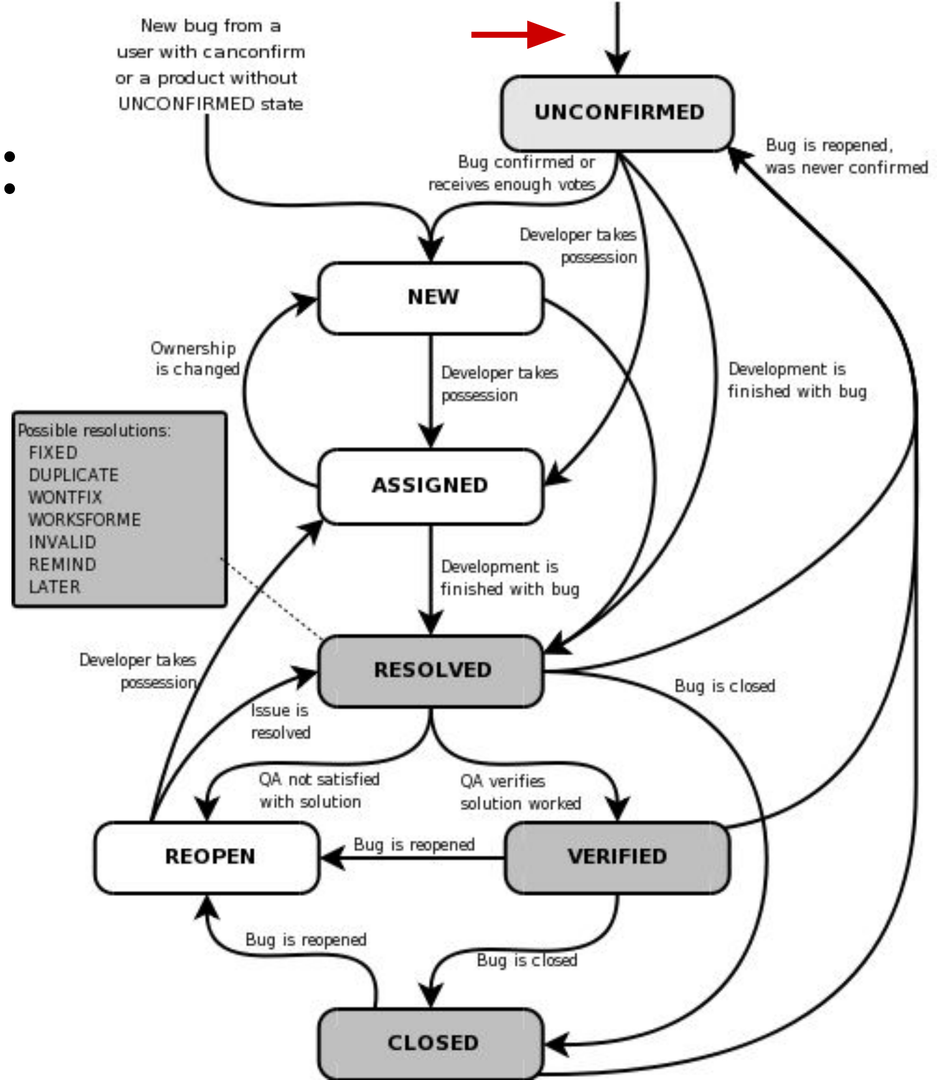
Defect report lifecycle: new bugs

- 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



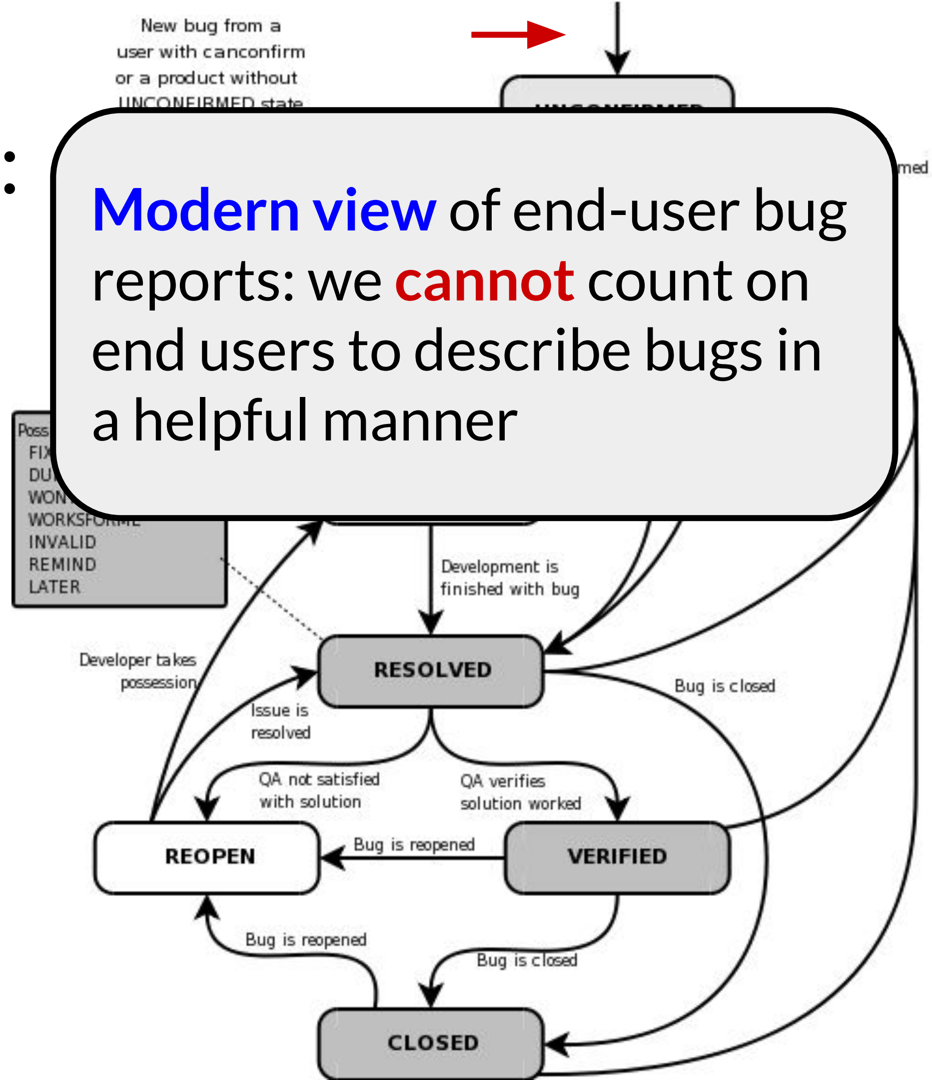
Defect report lifecycle: new bugs

- 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

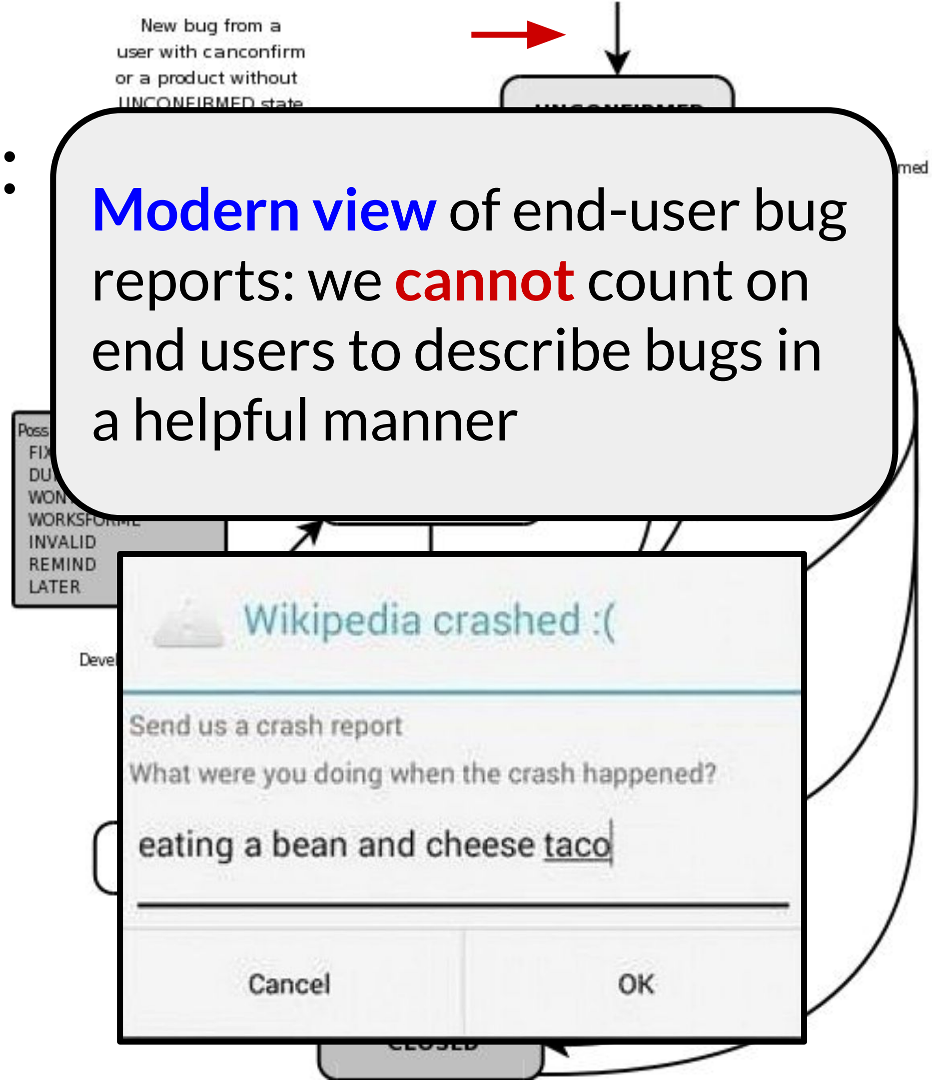


Modern view of end-user bug reports: we **cannot** count on end users to describe bugs in a helpful manner

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:
  - 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

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

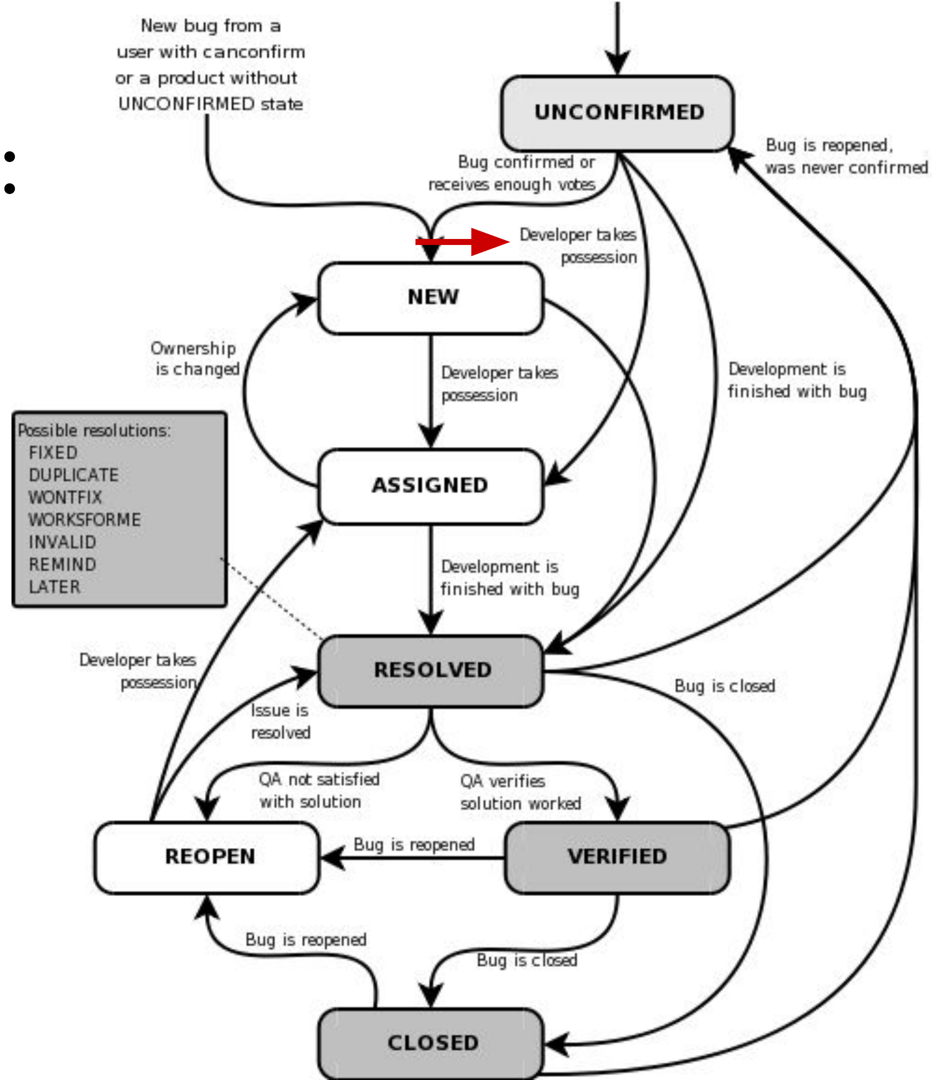
Defect reports: conversations

- 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

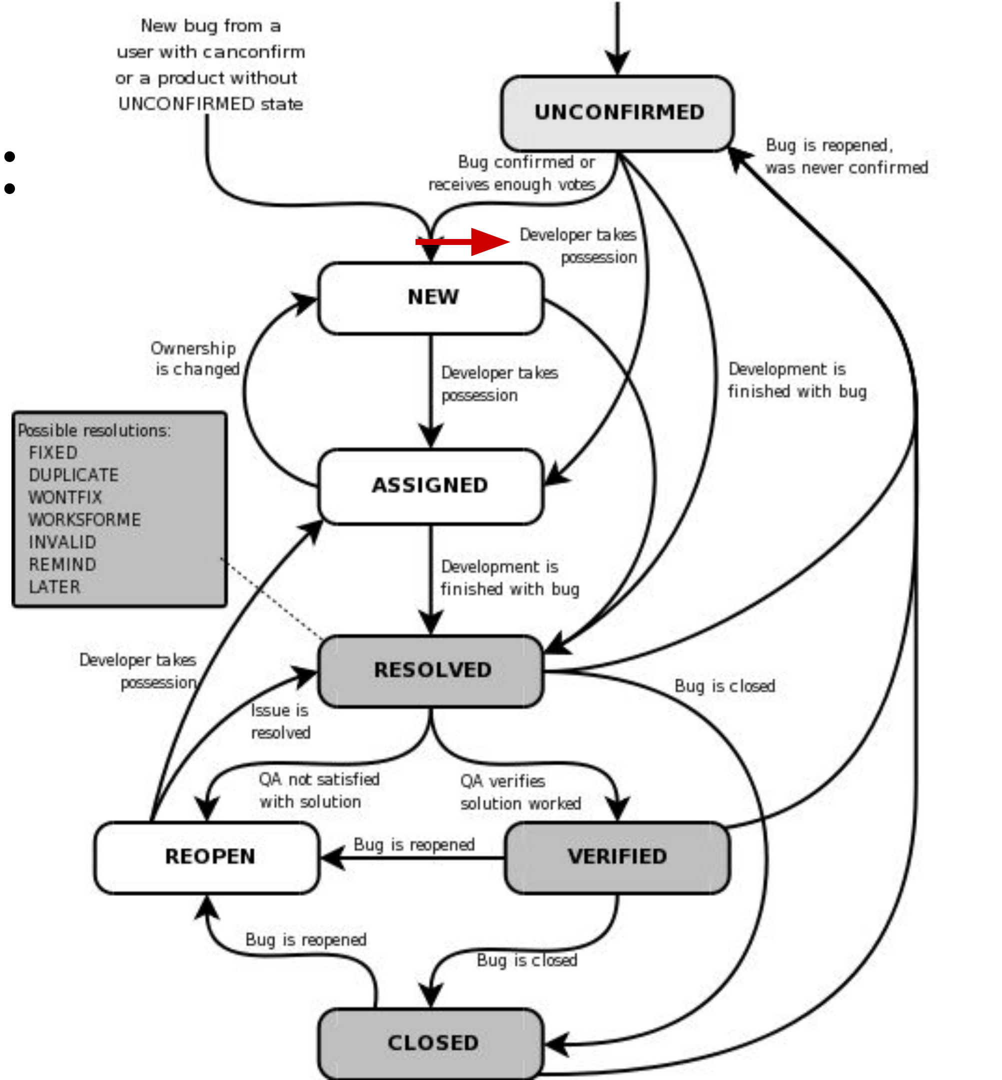
- Defect reports are **not static**
- Instead, they are **updated over time**
  - Request more info
  - Assign to a dev
  - Discuss solutions
- The report is a **log** of all relevant activity
- e.g.:
  - https://github.com/typetools/checker-framework/issues/4838
  - https://github.com/typetools/checker-framework/issues/3001

Defect report lifecycle: triage



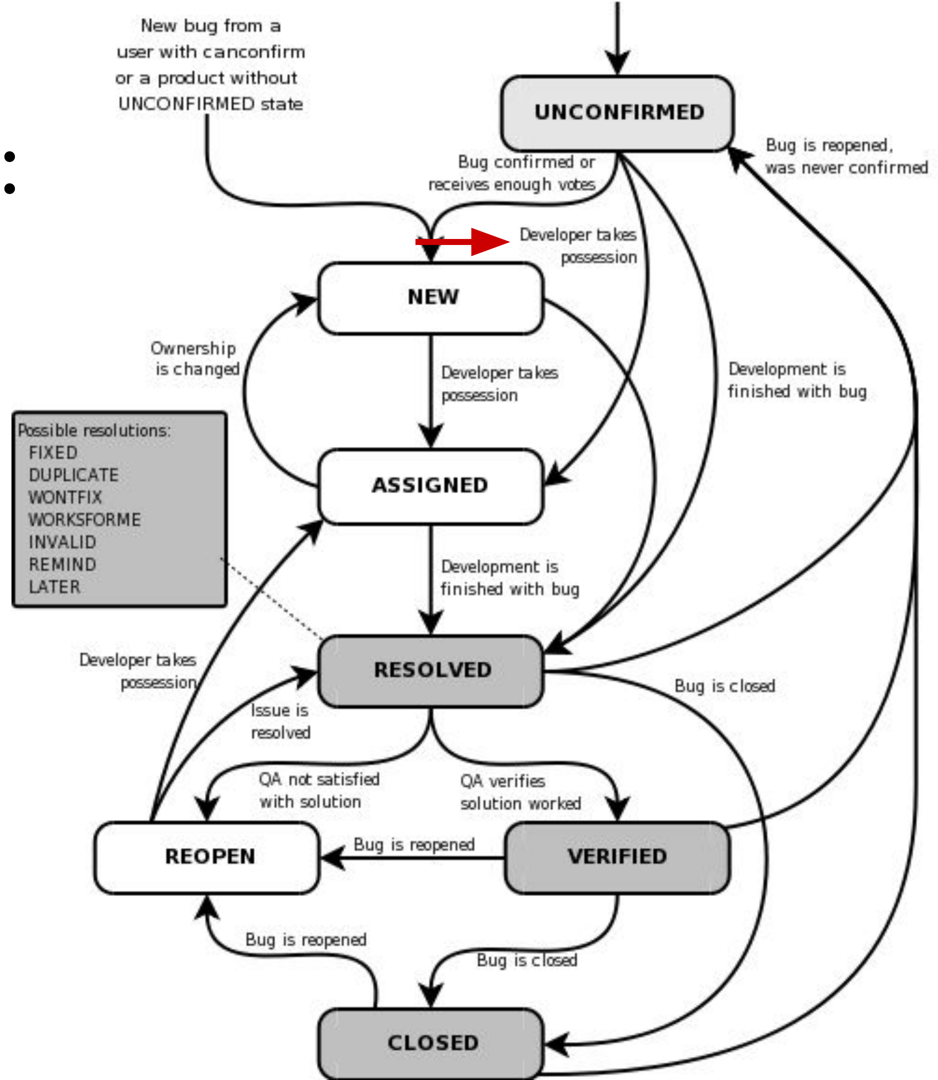
Defect report lifecycle: triage

- 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 differen
  - **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

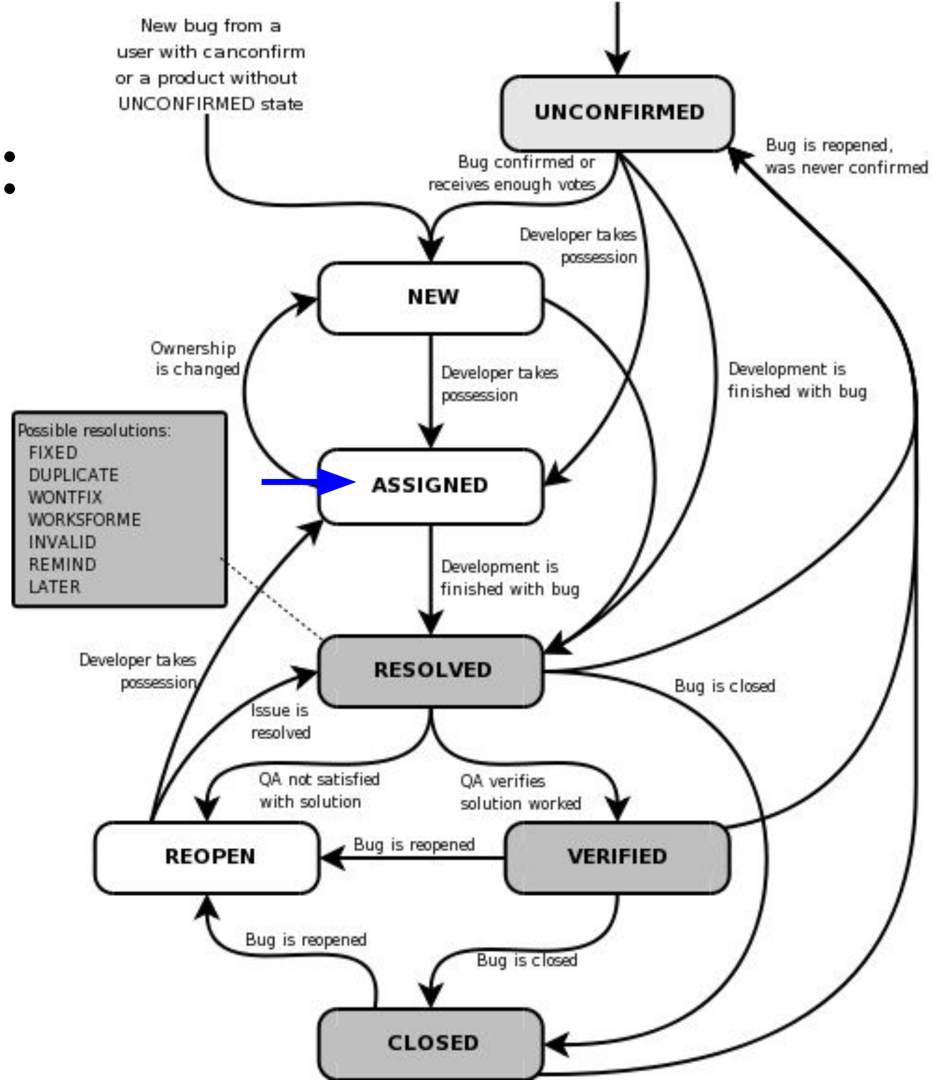
- related to, but officially different from, severity
  - **intuition**: if you have lots of high severity bugs, you need to prioritize between them
- severity and priority are used **together** (along with complexity, risk, etc.) to evaluate, prioritize and assign the resolution of reports

Defect report lifecycle: priority

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

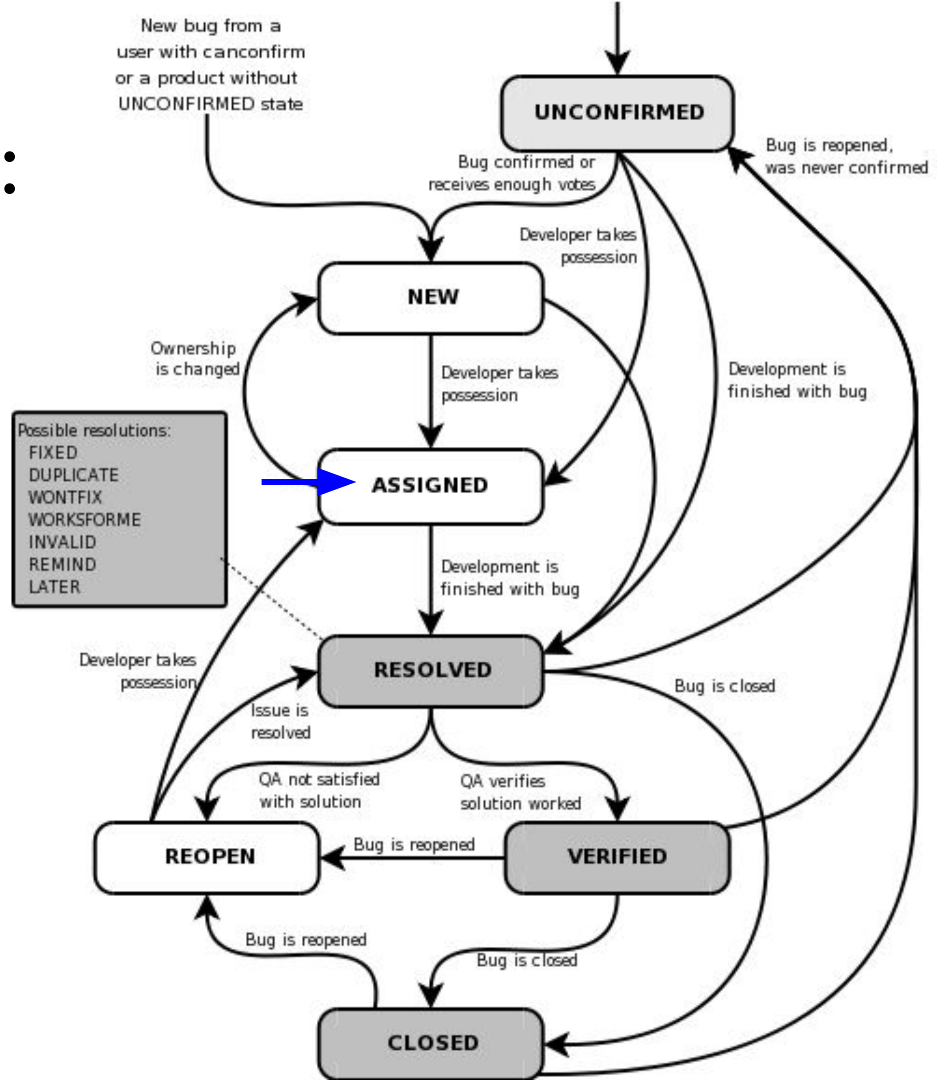
- related to, but officially different from, severity
  - **intuition**: if you have lots of high severity bugs, you need to prioritize between them
- severity and priority are used **together** (along with complexity, risk, etc.) to evaluate, prioritize and assign the resolution of reports
  - note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

Defect report lifecycle: assignment



Defect report lifecycle: assignment

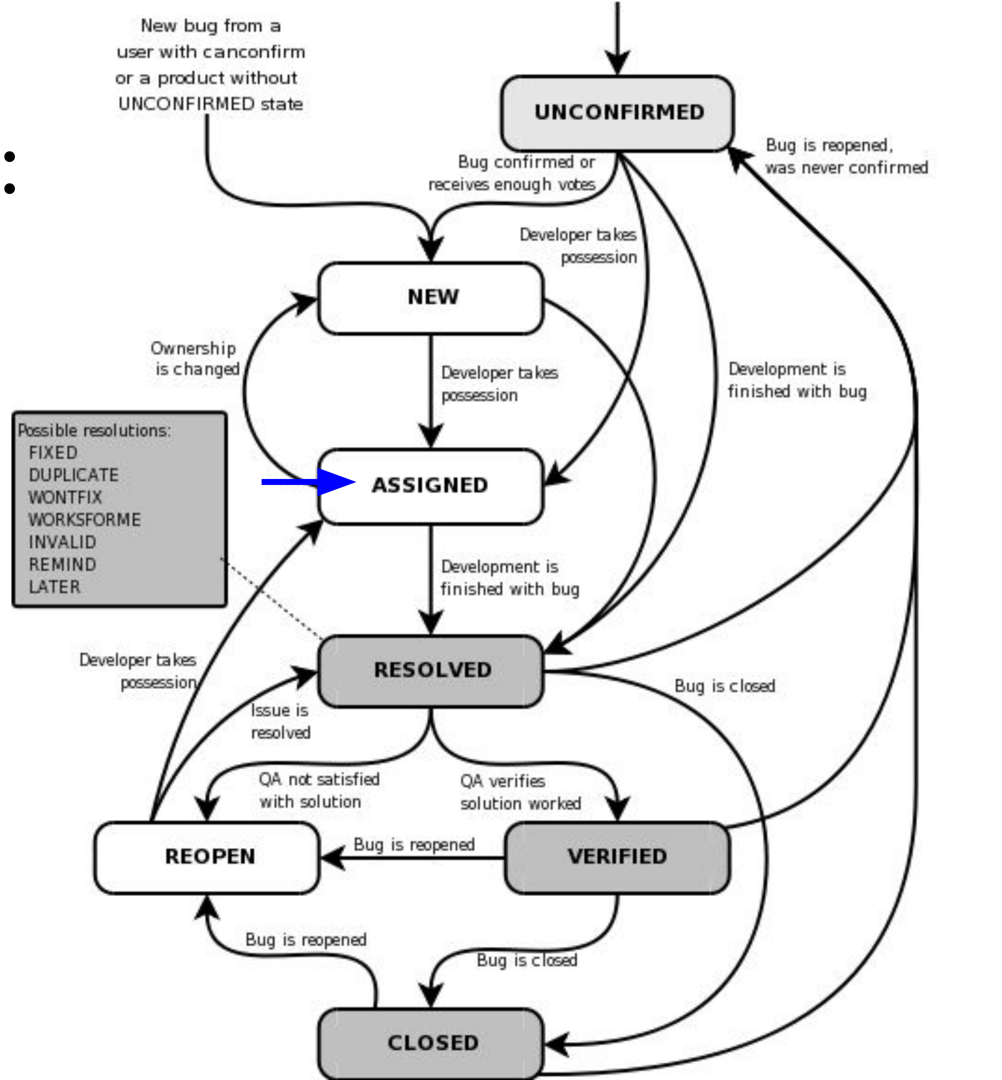
- Key question: **who** should fix this bug?



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

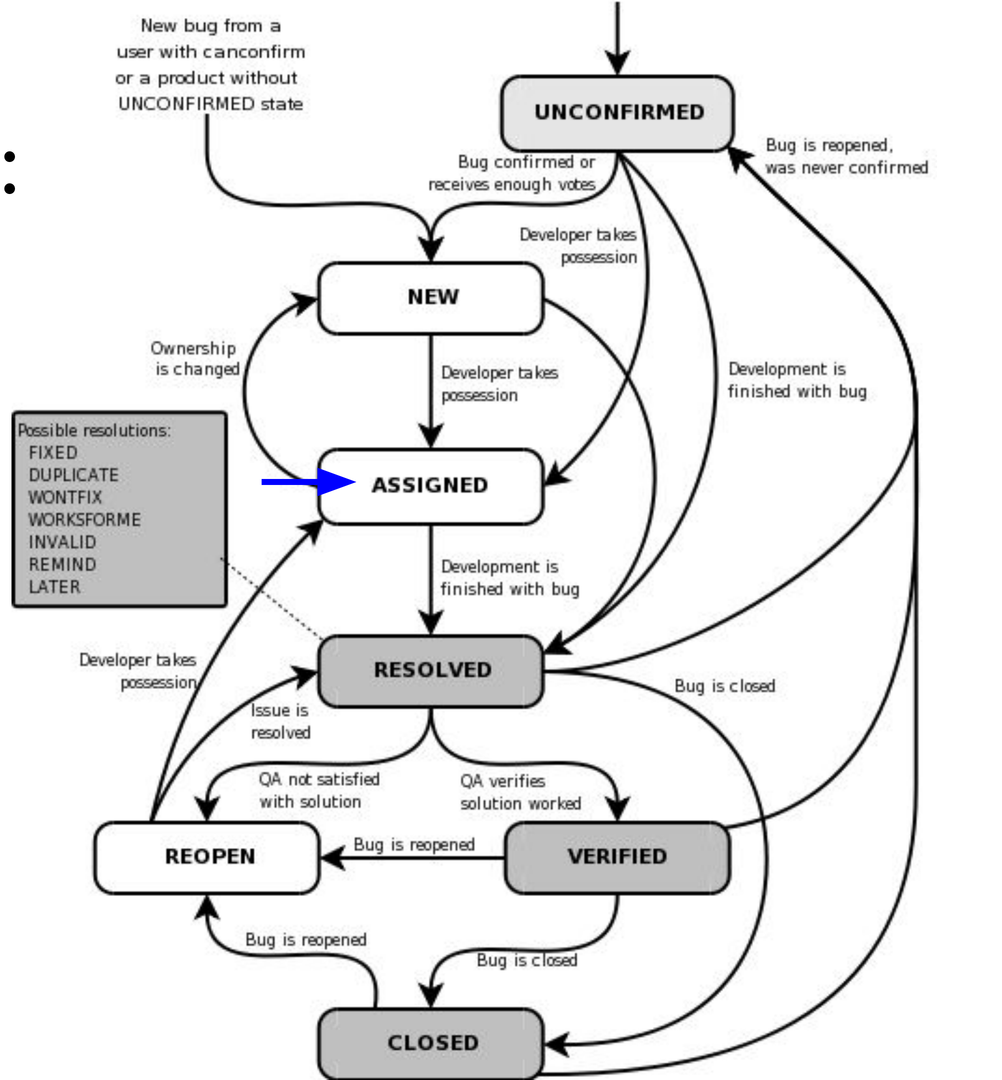


Defect report lifecycle: assignment

- 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”

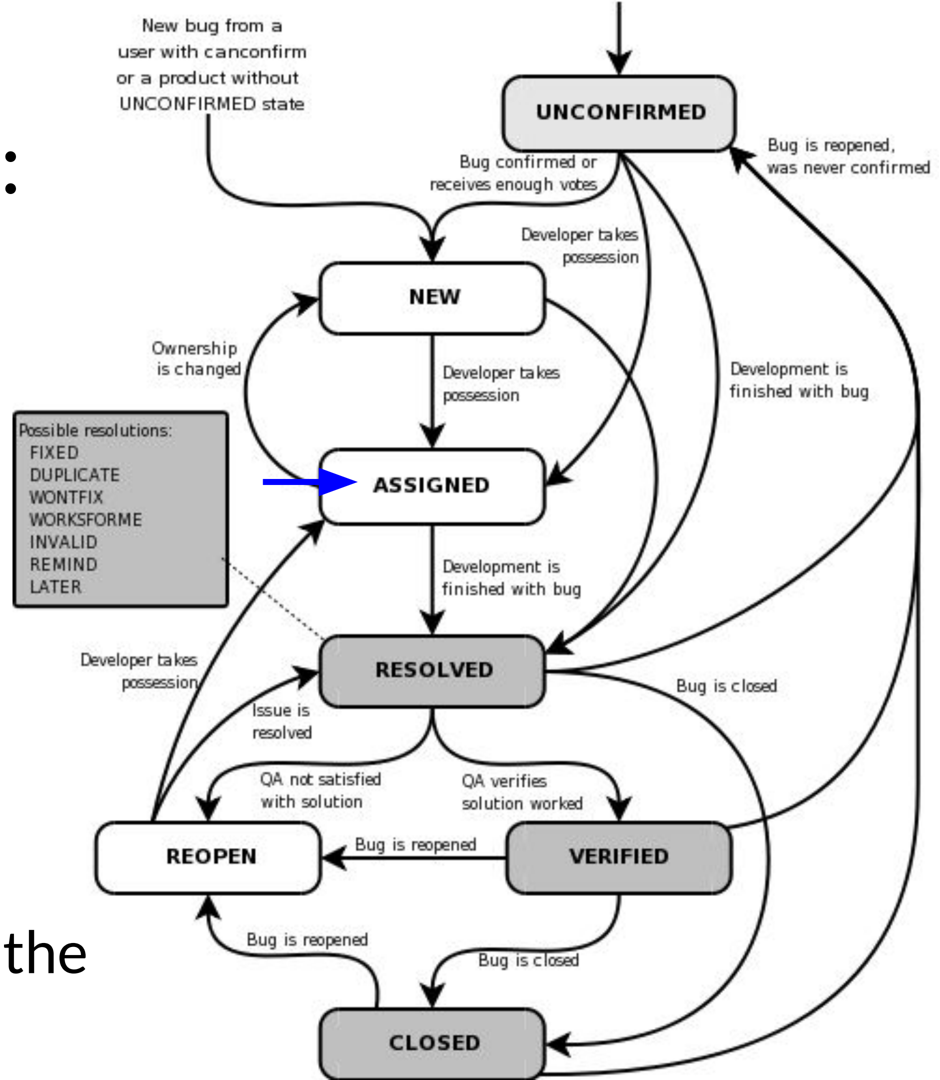


Defect report lifecycle: assignment

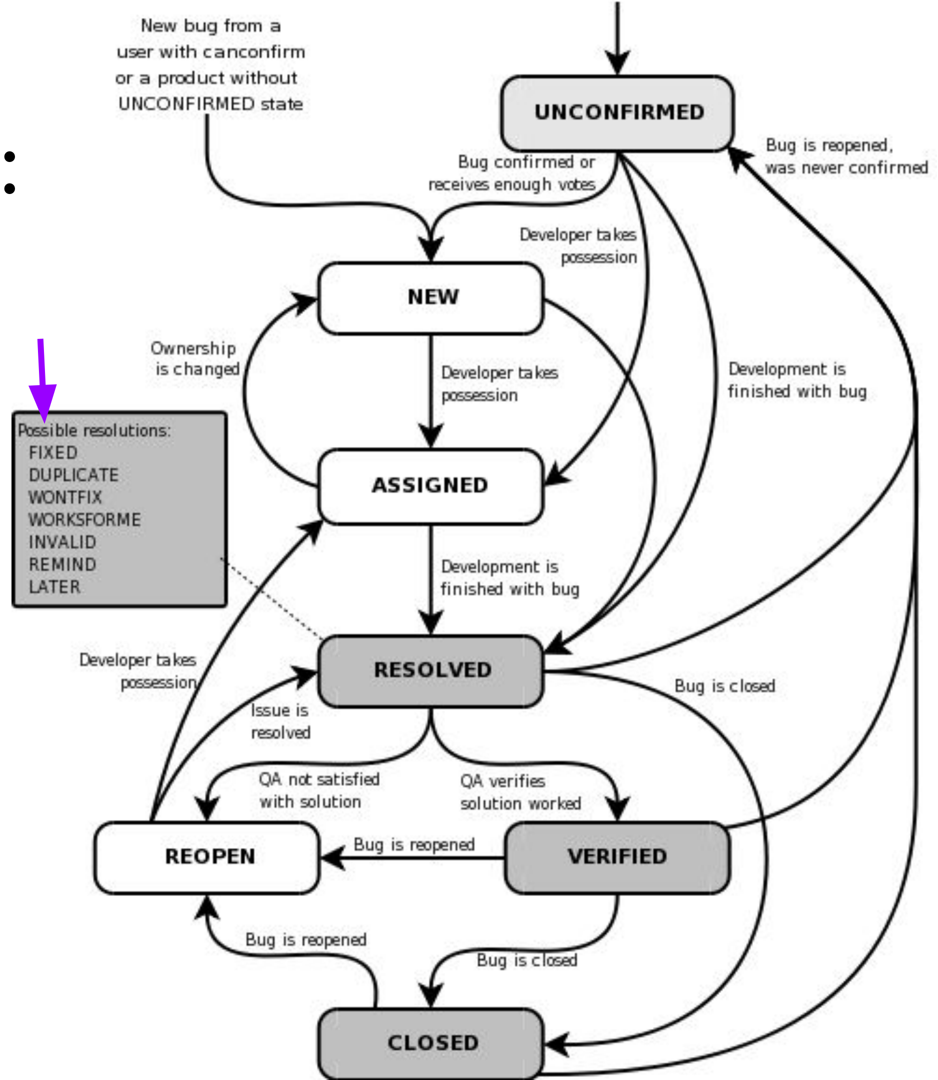
- Key question: **who** should fix this bug?

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

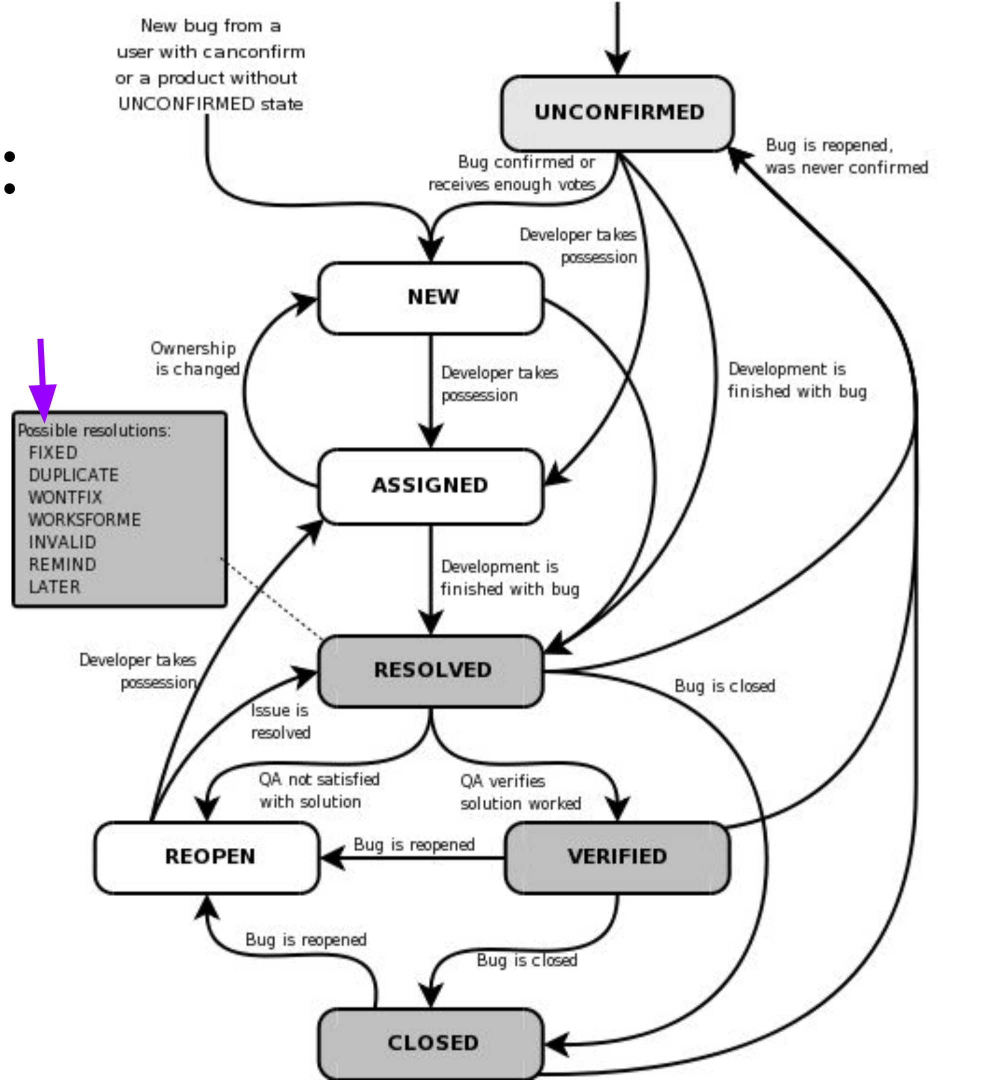


Defect report lifecycle: resolution



Defect report lifecycle: resolution

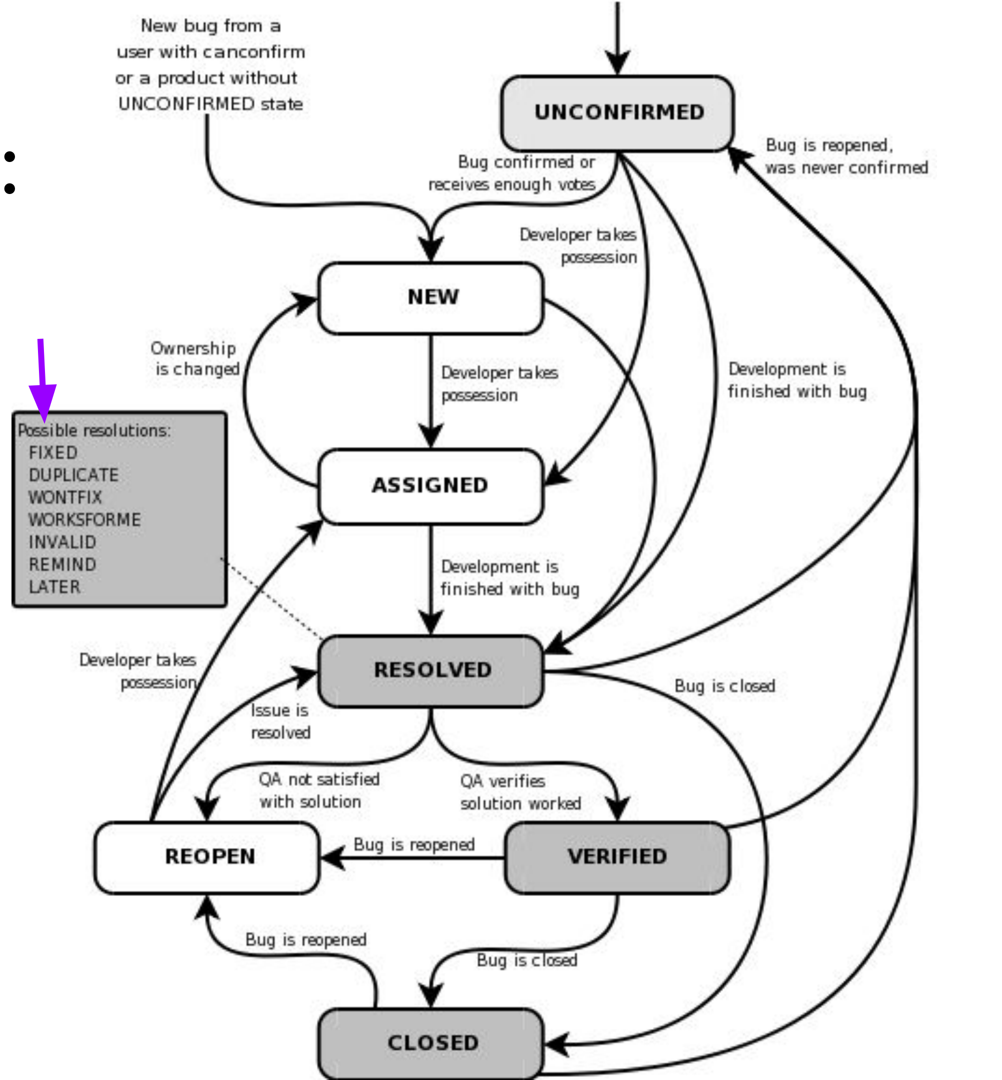
- Key question: did we **fix** it?



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

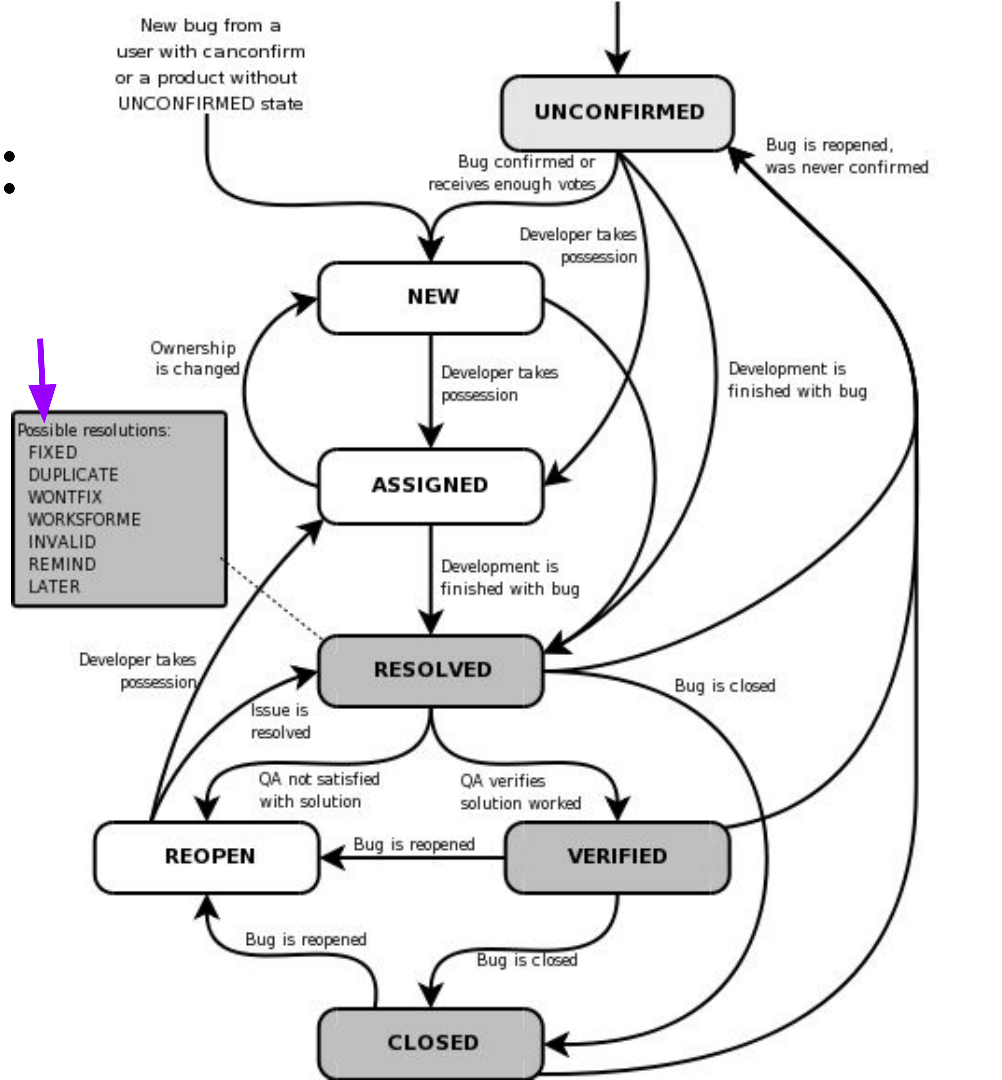


Defect report lifecycle: resolution

- 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)

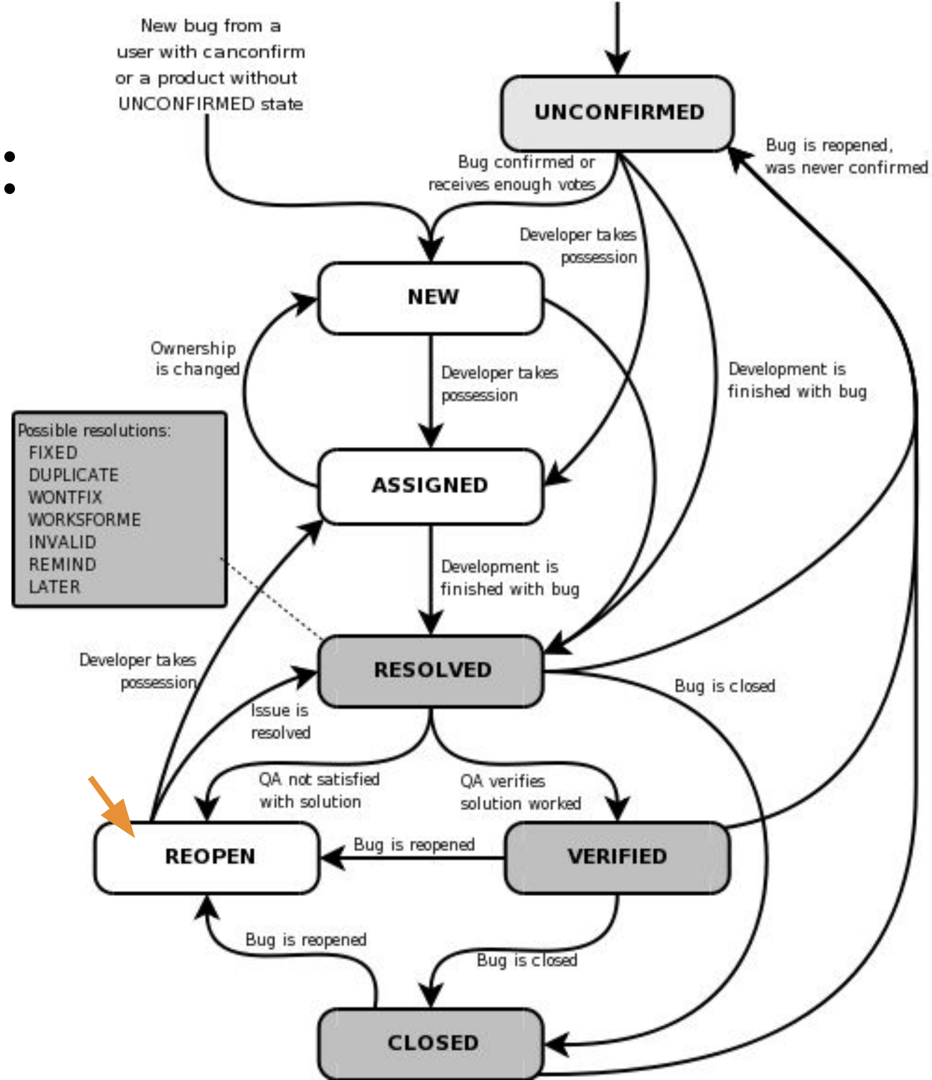
Thought question: what **fraction** of bug reports end up with each resolution?

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.

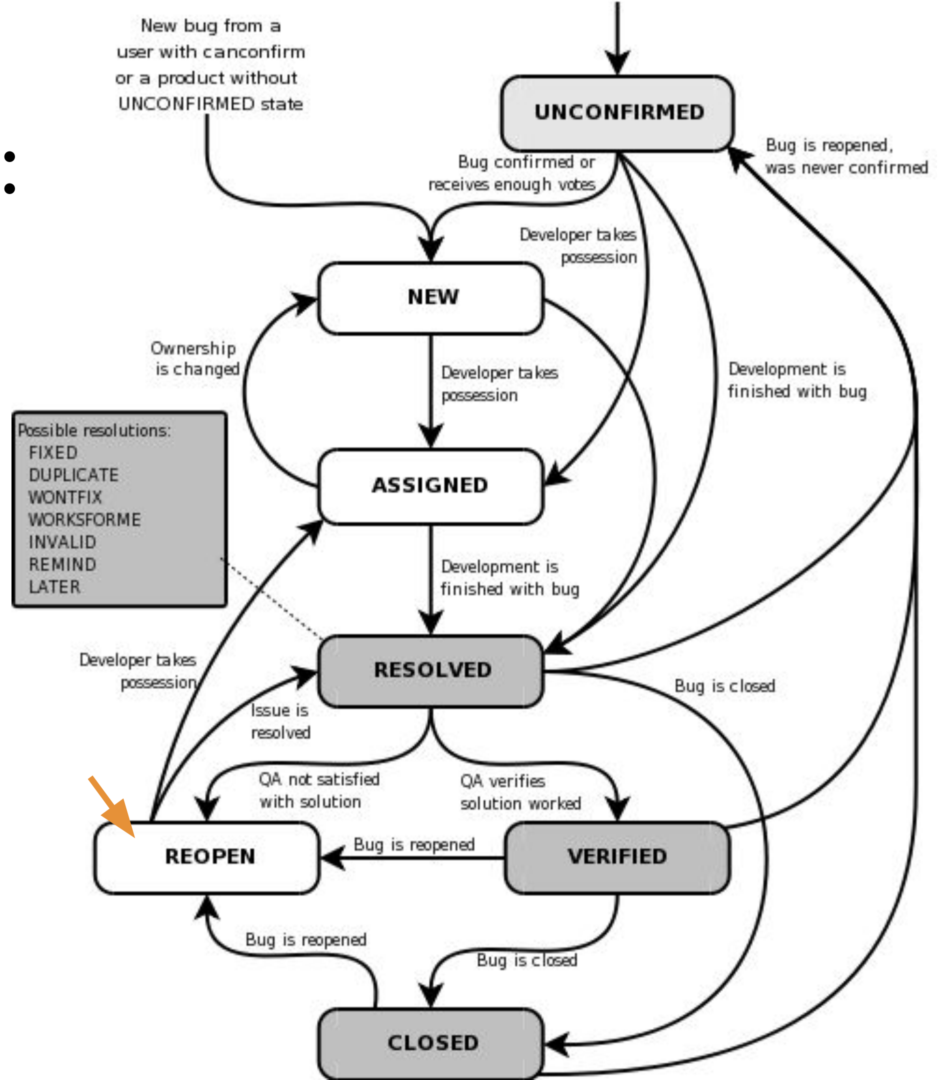
[Jalbert et al. Automated Duplicate Detection for Bug Tracking Systems. DSN 2008.]

Defect report lifecycle: reopening



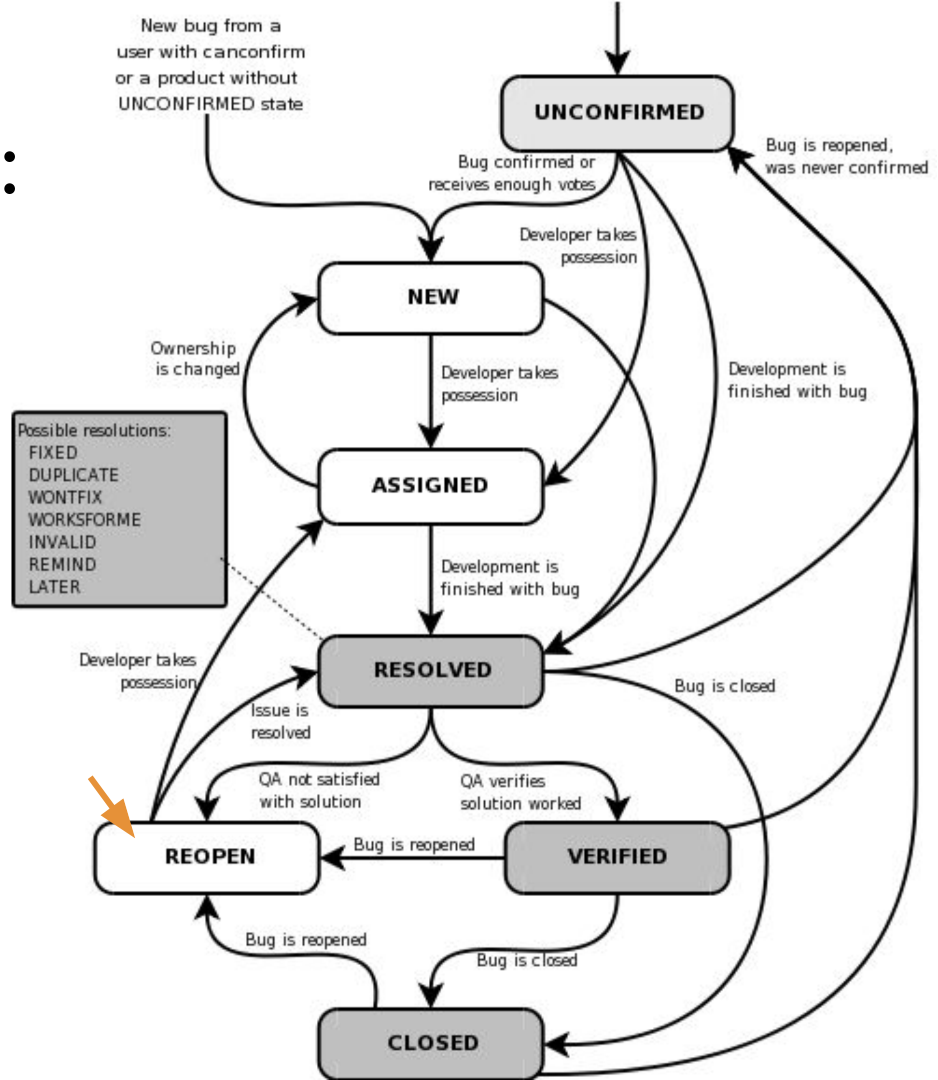
Defect report lifecycle: reopening

- 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.]

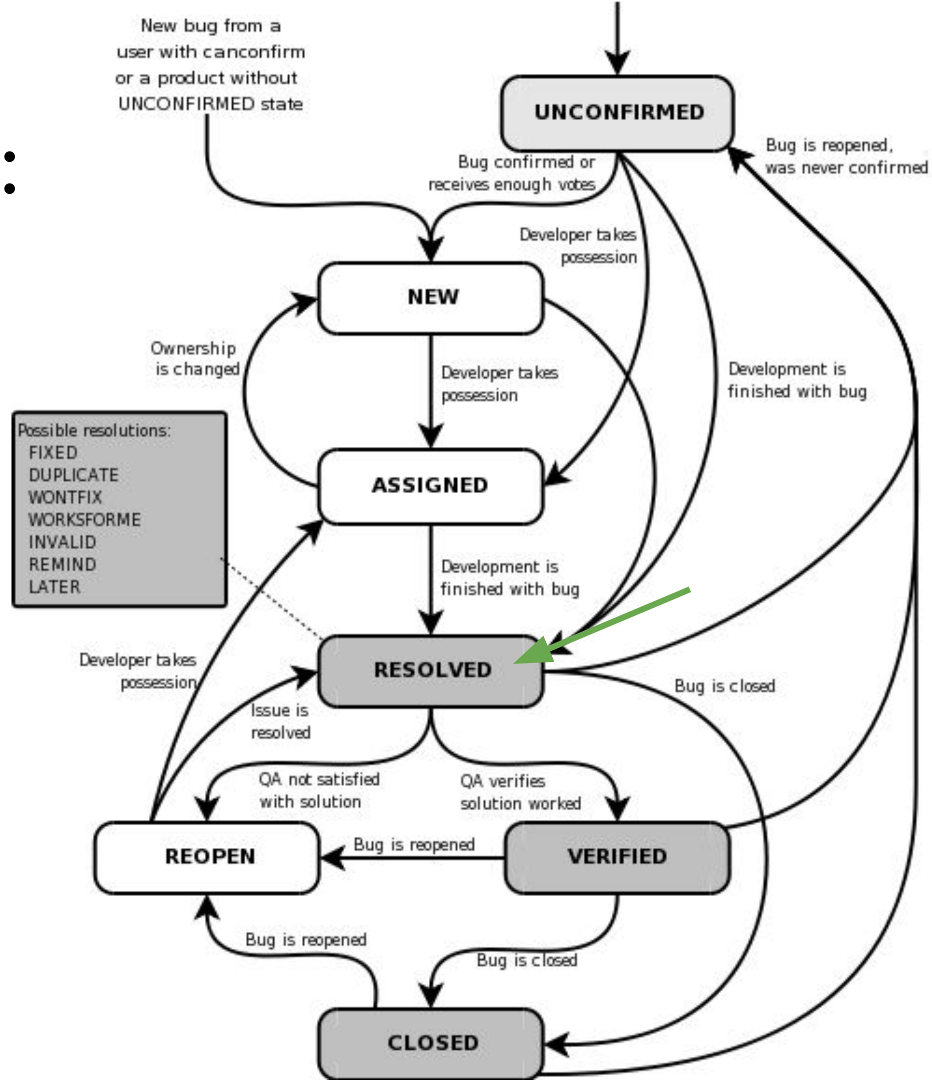
Defect report lifecycle: reopening

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**!

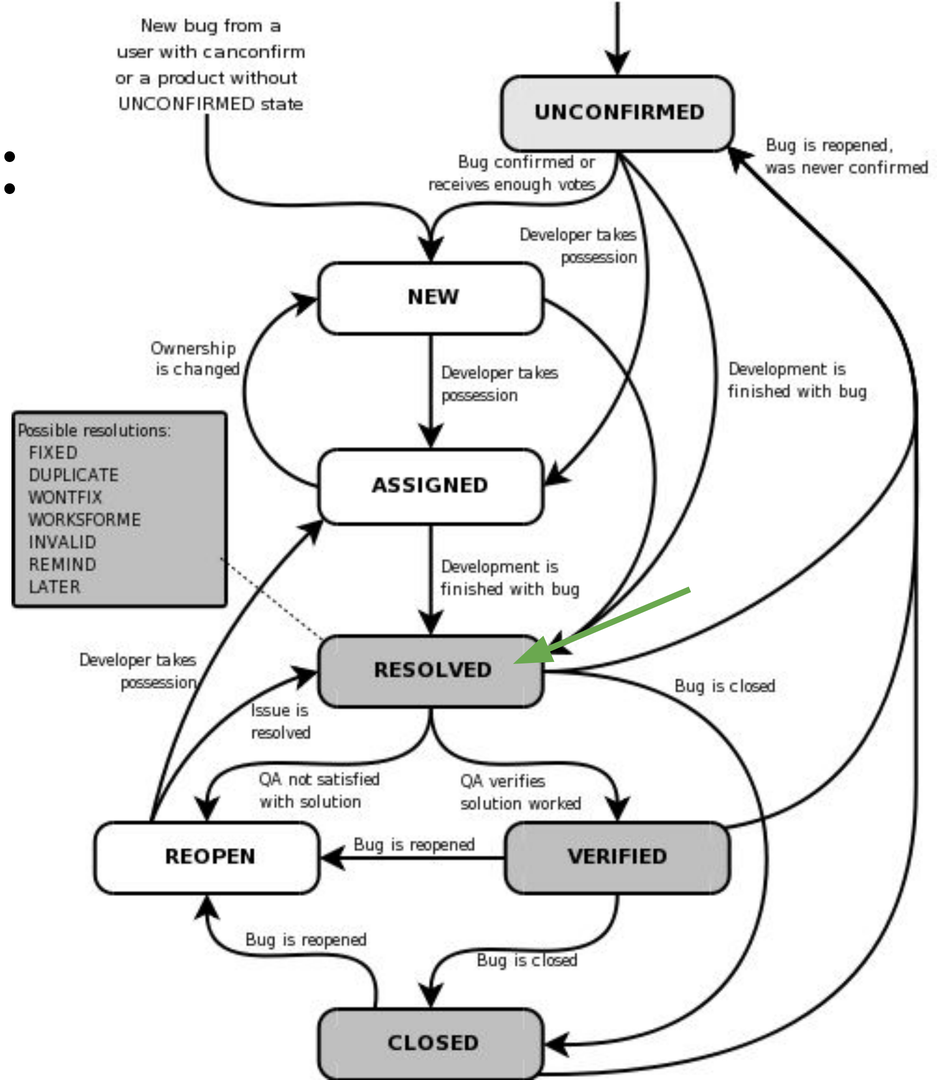
[Yin et al. How Do Fixes Become Bugs? ESEC/FSE 2011.]

Defect report lifecycle: fixing



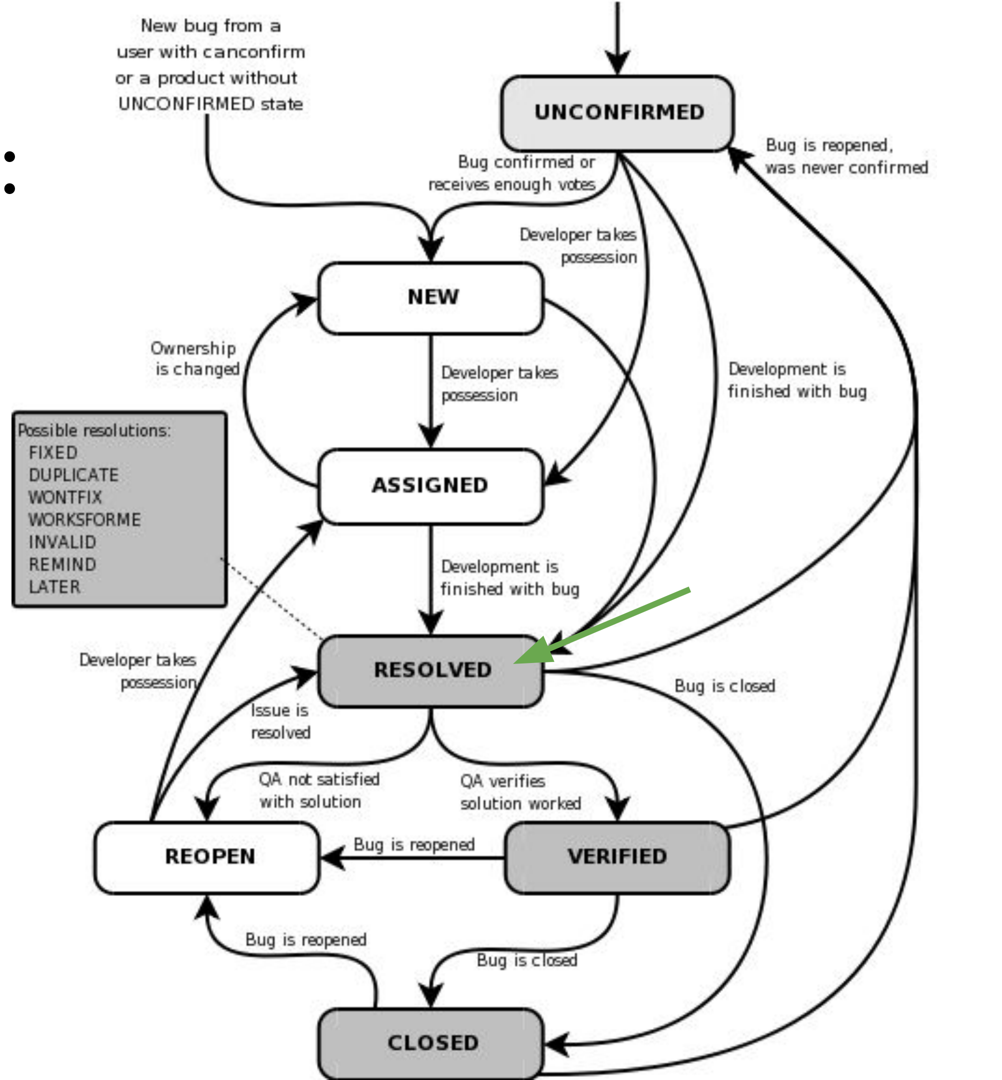
Defect report lifecycle: fixing

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



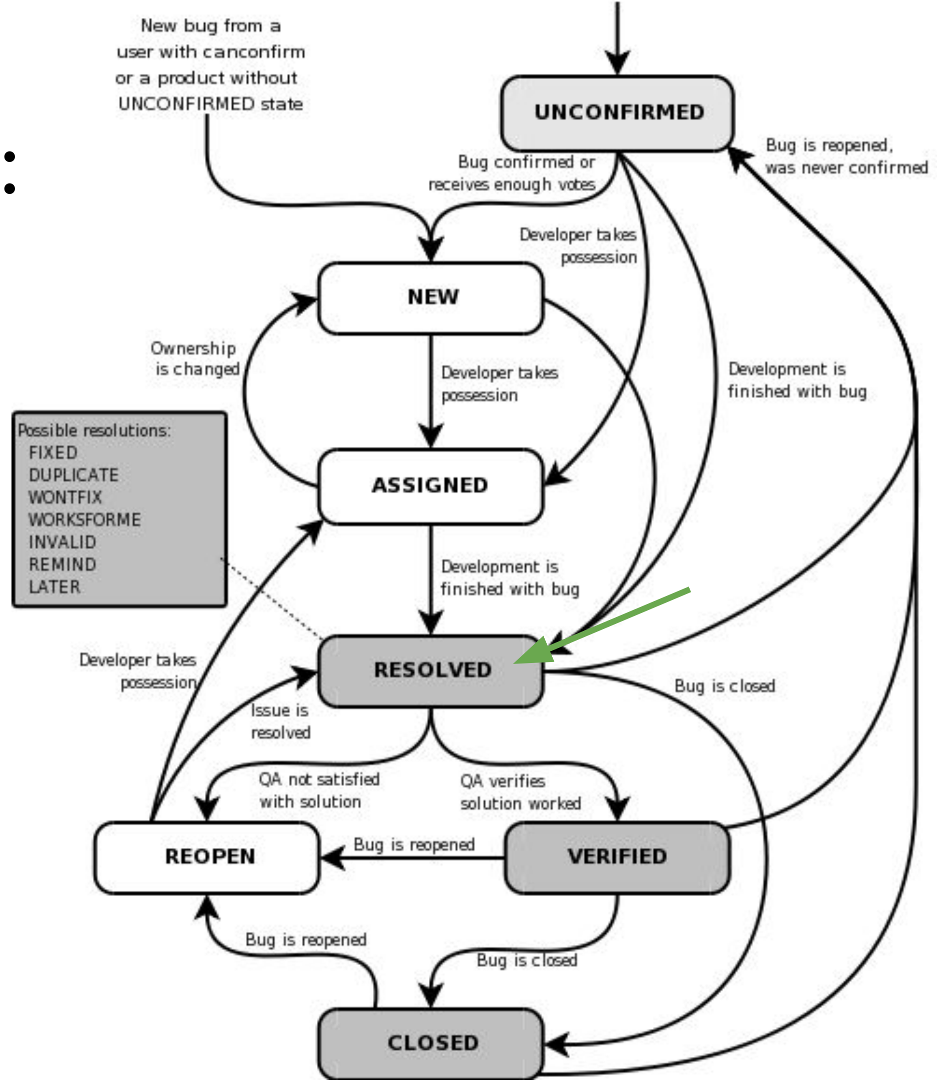
Defect report lifecycle: fixing

- 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 Thursday's lecture on debugging



Debugging (Part 1/2)

Today's agenda:

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

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!

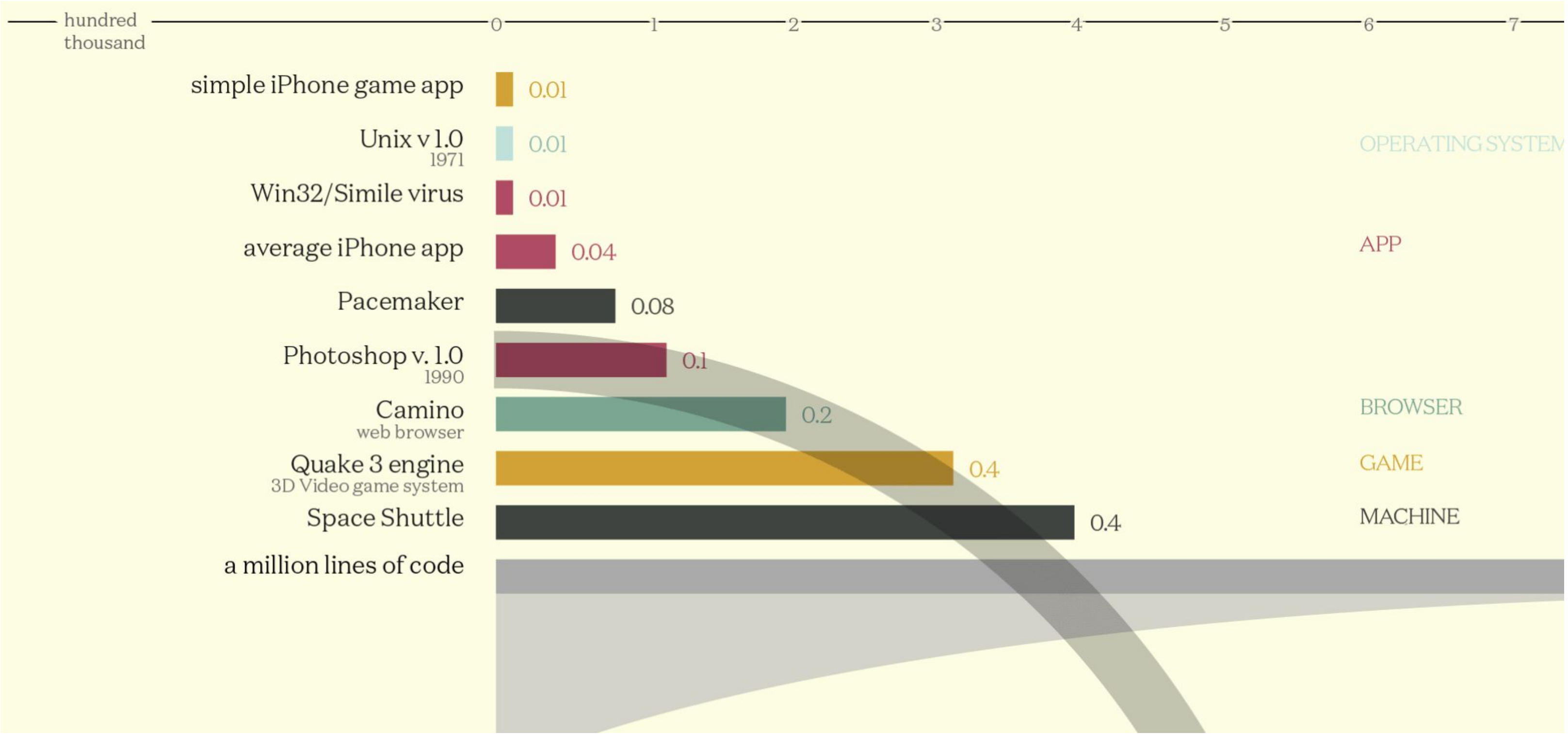
Debugging: what makes it difficult?

- 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?

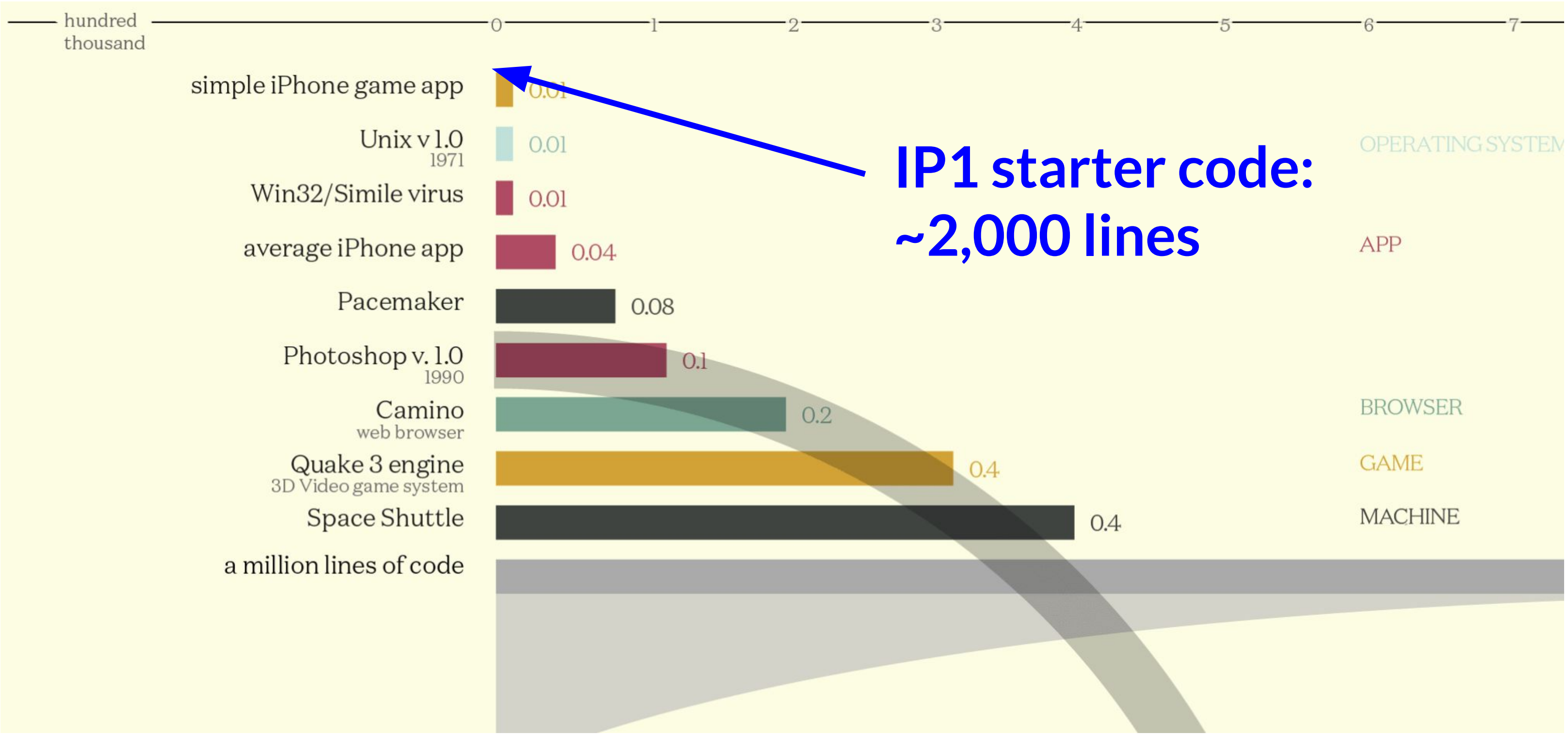
- modern software is **unimaginably huge**
  - analogy: scale of space vs human scale
    - "Space is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space." – Douglas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



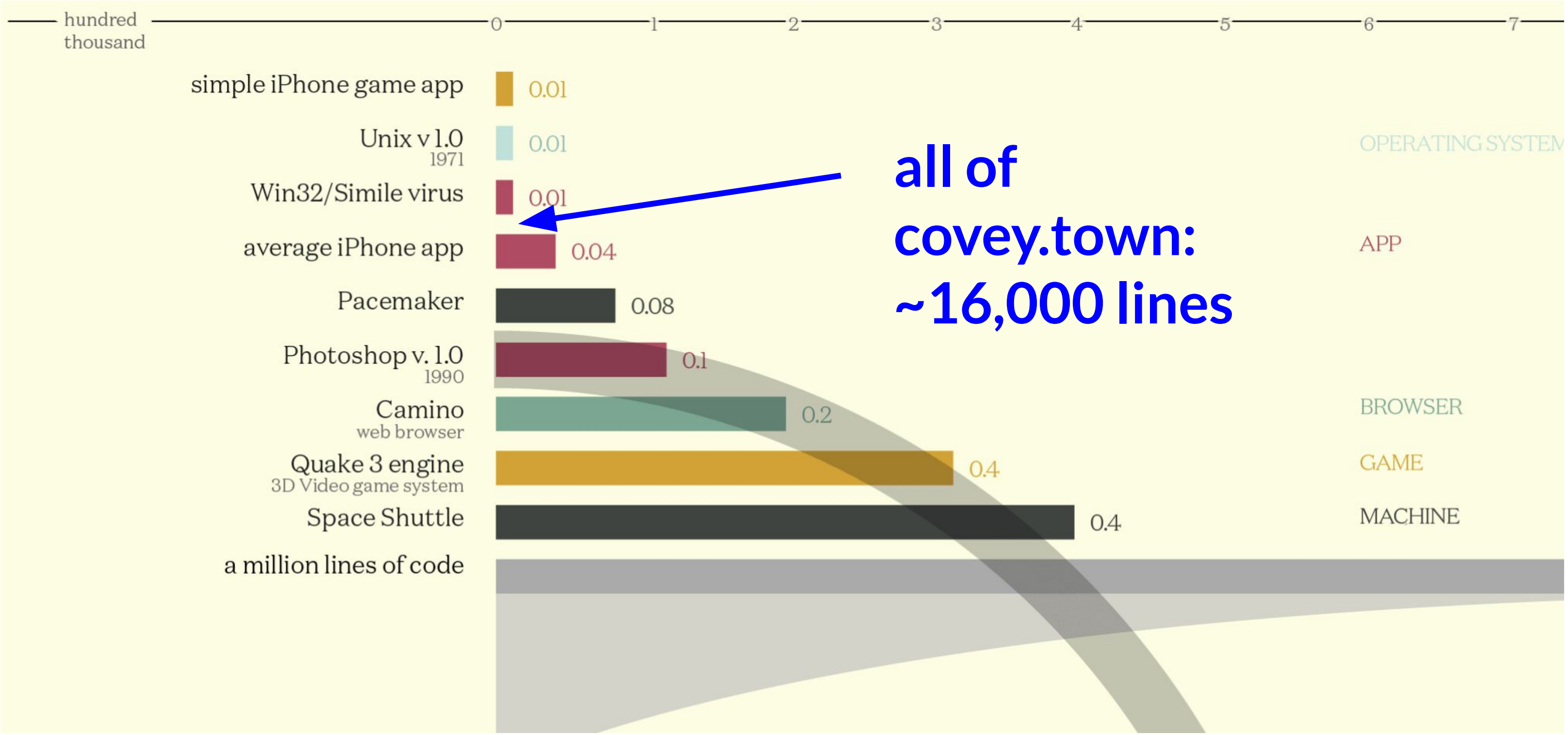
https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?



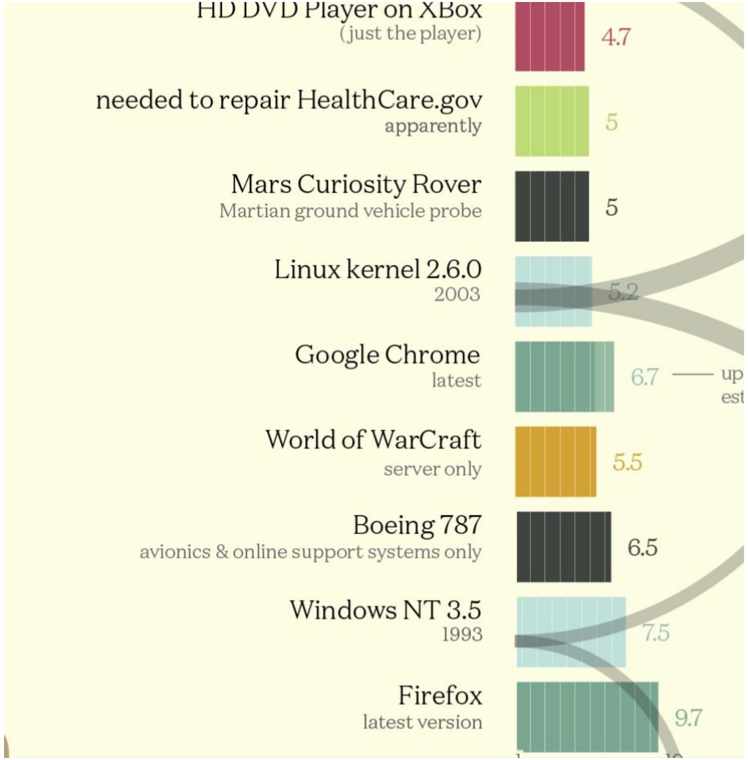
https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?



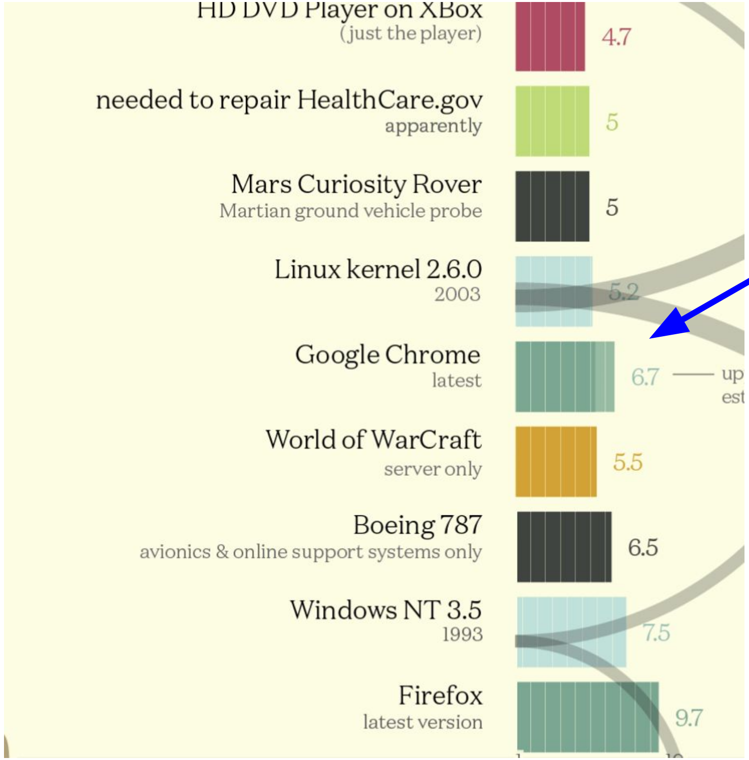
https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?



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

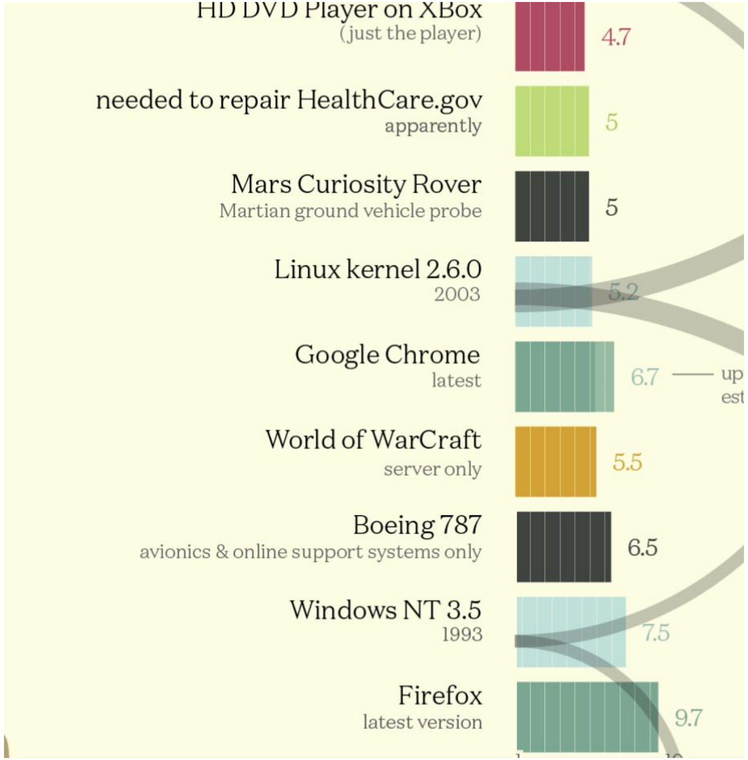
How big are programs, really?

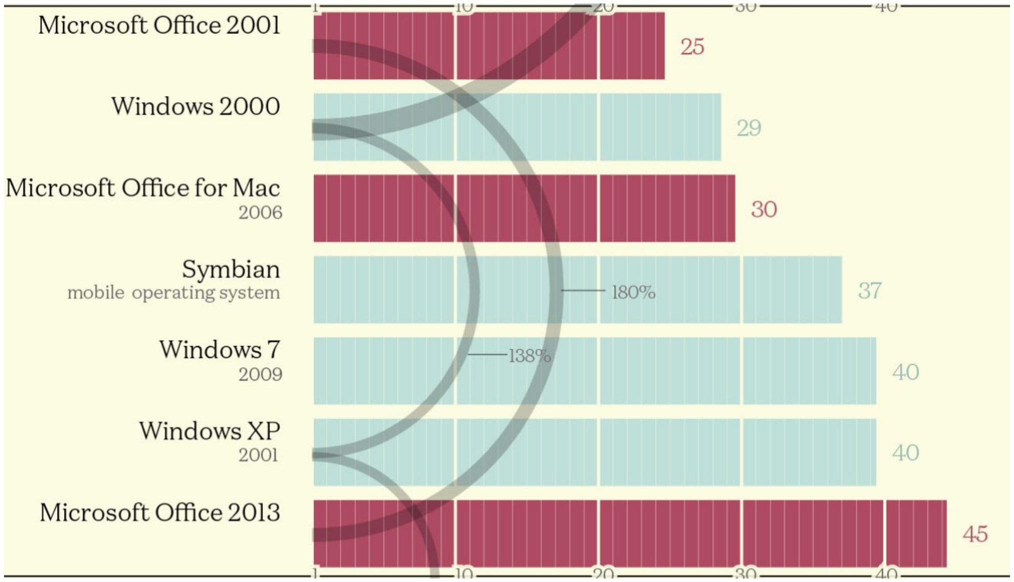


Chrome at ~7M LoC is ~400x bigger than covey.town

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

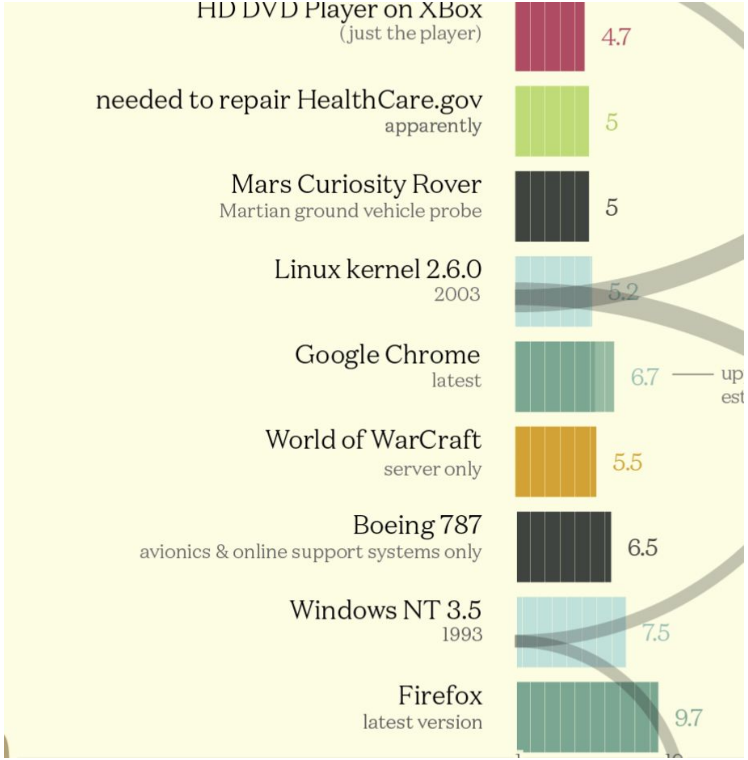
How big are programs, really?



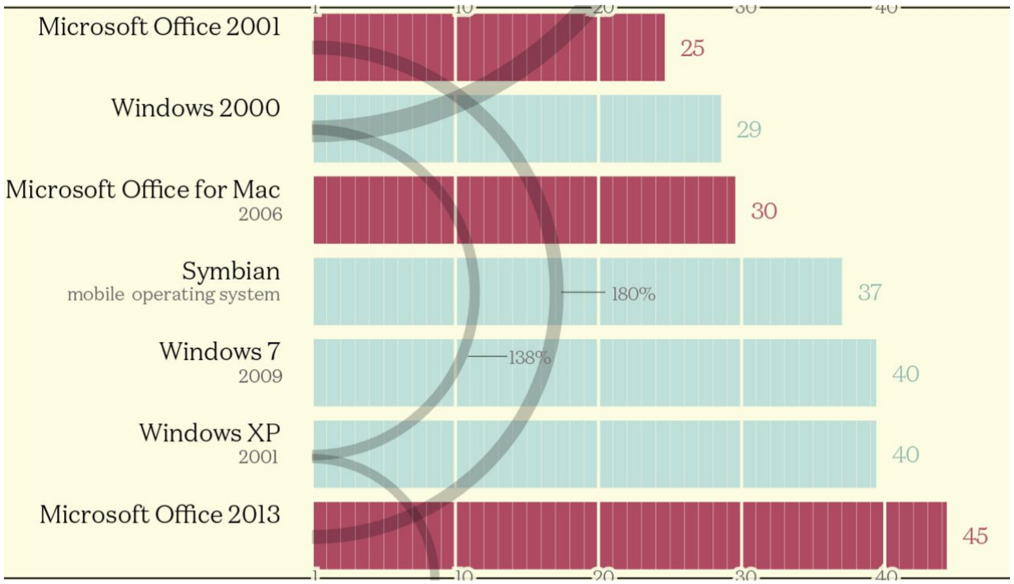


https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

How big are programs, really?

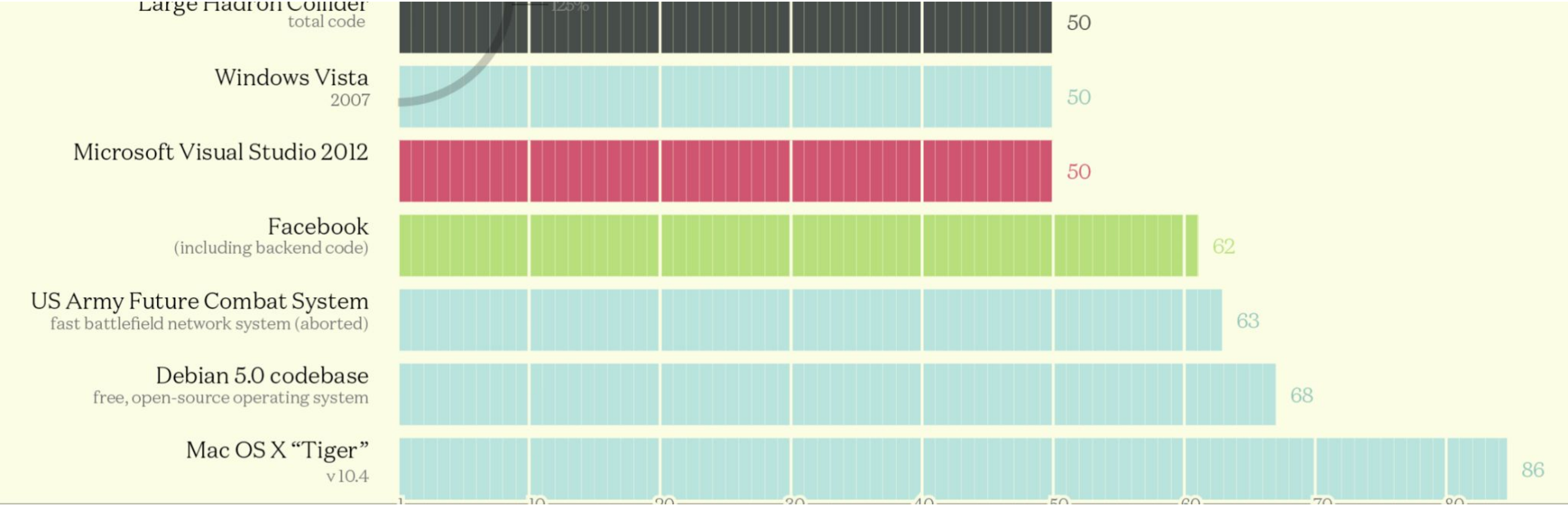


Chrome is small compared to even old versions of Windows!



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

CADE METZ BUSINESS 09.16.15 10:00 AM

SHARE

SHARE

TWEET

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
  - debuggers
  - delta debugging

Debugging (Part 2/2)

Today's agenda:

- **Reading quiz**
- Debugging
    - printf debugging and logging
    - debuggers
    - delta debugging

Reading quiz: debugging (part 2)

Q1: **TRUE** or **FALSE**: delta debugging requires a test to prove that each circumstance is really failure inducing

Q2: Which of the following tasks did the article's authors use delta debugging to improve:

A. isolating differences
B. finding failure-inducing code changes
C. simplifying user interactions
D. all of the above

Reading quiz: debugging (part 2)

Q1: **TRUE** or **FALSE**: delta debugging requires a test to prove that each circumstance is really failure inducing

Q2: Which of the following tasks did the article's authors use delta debugging to improve:

- **A.** isolating differences
- **B.** finding failure-inducing code changes
- **C.** simplifying user interactions
- **D.** all of the above

Reading quiz: debugging (part 2)

Q1: **TRUE** or **FALSE**: delta debugging requires a test to prove that each circumstance is really failure inducing

Q2: Which of the following tasks did the article's authors use delta debugging to improve:

A. isolating differences
B. finding failure-inducing code changes
C. simplifying user interactions
D. all of the above

Debugging (Part 2/2)

Today's agenda:

- Reading quiz
- **Debugging**
  - printf debugging and logging
  - debuggers
  - delta debugging

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 guess
  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

Debugging strategies

- "printf" debugging: using print statements to find a bug
  - and its larger-scale cousin: **logging**
- debuggers: **inspecting program state** while it is running
  - we'll talk a little about how they work
- delta debugging
  - a **formalization** of the scientific approach to debugging

Debugging (Part 2/2)

Today's agenda:

- Reading quiz
- Debugging
  - **printf debugging and logging**
  - debuggers
  - delta debugging

“printf” debugging

- probably your most common debugging strategy already!

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a particular point

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a particular point
- advantages:
  - easy and natural

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a particular point
- advantages:
    - easy and natural
- disadvantages:
    - must recompile, rerun program each time you want to test something else
    - sometimes considered “unprofessional”

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a part
- advantages:
    - easy and natural
- disadvantages:
    - must recompile, re
      something else
    - sometimes considered “unprofessional”

This is a **misconception**: professional engineers commonly use printf debugging. But printf debugging should be just one tool in your toolbox of debugging strategies!

Logging

Definition: ***logging*** is the process of recording information about the program's internal state as it runs via a printf-like interface

Logging

Definition: ***logging*** is the process of recording information about the program's internal state as it runs via a printf-like interface

- logging is a key technology for **monitoring** modern systems
  - e.g., via tools like Log4j, slf4j, etc.

Logging

Definition: *logging* is the process of recording information about the program's internal state as it runs via a printf-like interface

- logging is a key technology for **monitoring** modern systems
  - e.g., via tools like Log4j, slf4j, etc.
- logs also play a major role in debugging **large-scale failures** of important distributed systems

Logging

Definition: ***logging*** is the process of recording information about the program's internal state as it runs via a printf-like interface

- logging is a key technology for **monitoring** modern systems
  - e.g., via tools like Log4j, slf4j, etc.
- logs also play a major role in debugging **large-scale failures** of important distributed systems
  - we'll discuss this more when we talk about **post-mortems** in our DevOps lectures, near the end of the semester

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

the log itself is usually a static field; the logging framework instantiates it, etc.

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

"debug" means if debug-level logging isn't enabled in the framework, this becomes a no-op

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

"debug" means if debug-level logging isn't enabled in the framework, this becomes a no-op

levels:
error ⊆ warning ⊆ info ⊆ debug

developer chooses one level, all lower level messages are also logged

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

printf-like syntax isn't just for show: goal here is lazy evaluation, so that if debug logging isn't enabled, this string is never constructed

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug(“myVariable=%s”, myVariable);
```

arguments to printf passed by reference, so if debug-level logging is off, this argument’s toString() method is never called

Logging: advice

Logging: advice

- **Do** log lots of information at debug or info level, so that if something is wrong with your service you can quickly get lots of information that you can use to debug it.

Logging: advice

- **Do** log lots of information at debug or info level, so that if something is wrong with your service you can quickly get lots of information that you can use to debug it.
- **Don't** log sensitive data (e.g., credit card numbers in plaintext!)
  - this is a surprisingly common and important problem - developers have a tendency to log anything that might be useful when debugging a failure later!

Debugging (Part 2/2)

Today's agenda:

- Reading quiz
- Debugging
  - printf debugging and logging
  - **debuggers**
  - delta debugging

Debuggers

Debuggers

Definition: a ***debugger*** is "a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables." [definition from Microsoft Developer Network]

Debuggers

Definition: a ***debugger*** is "a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables." [definition from Microsoft Developer Network]

- Can operate on source code or assembly code

Debuggers

Definition: a ***debugger*** is "a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables." [definition from Microsoft Developer Network]

- Can operate on source code or assembly code
- **Inspect** the values of registers, memory

Debuggers

Definition: a ***debugger*** is "a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables." [definition from Microsoft Developer Network]

- Can operate on source code or assembly code
- **Inspect** the values of registers, memory
- **Key Features** (we'll explain all of them): attach to process, single-stepping, breakpoints, conditional breakpoints, watchpoints

Debuggers: how do they work

Debuggers: how do they work: signals

Debuggers: how do they work: signals

- A *signal* is an **asynchronous** notification sent to a process about an event:
  - User pressed Ctrl-C (or did kill %pid)
    - Or asked the Windows Task Manager to terminate it
  - Exceptions (divide by zero, null pointer)
  - From the OS (SIGPIPE)

Debuggers: how do they work: signals

- A *signal* is an **asynchronous** notification sent to a process about an event:
  - User pressed Ctrl-C (or did kill %pid)
    - Or asked the Windows Task Manager to terminate it
  - Exceptions (divide by zero, null pointer)
  - From the OS (SIGPIPE)
- You can install a **signal handler** – a procedure that will be executed when the signal occurs.

Debuggers: how do they work: signals

- A *signal* is an **asynchronous** notification sent to a process about an event:
    - User pressed Ctrl-C (or did kill %pid)
        - Or asked the Windows Task Manager to terminate it
    - Exceptions (divide by zero, null pointer)
    - From the OS (SIGPIPE)
- You can install a **signal handler** – a procedure that will be executed when the signal occurs.
    - Signal handlers are vulnerable to **race conditions**. Why?

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support
- There is a **special system call** that allows one process to act as a debugger for a target

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support
- There is a **special system call** that allows one process to act as a debugger for a target
  - What are the **security** concerns?

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support
- There is a **special system call** that allows one process to act as a debugger for a target
  - What are the **security** concerns?
- Once this is done, the debugger can basically "**catch signals**" delivered to the target
  - this isn't exactly what happens, but it's a good explanation ...

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:

A ***breakpoint*** is a user-specified program statement on which the debugger should stop the program and begin an interactive debugging session

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
    - Attach to target

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
    - Attach to target
    - Set up signal handler

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
    - Attach to target
    - Set up signal handler
    - Add in **exception causing instructions** at desired breakpoints

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
  - Attach to target
  - Set up signal handler
  - Add in **exception causing instructions** at desired breakpoints
  - **Inspect** globals, do other debugger things, etc.

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf(“debugger prompt: \n”);
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

All code added by the debugger in **purple**

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf("debugger prompt: \n");
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

"**BREAKPOINT**" macro is guaranteed to cause SIGSEGV

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf(“debugger prompt: \n”);
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

debugger **registers** a **SIGSEGV handler**

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf(“debugger prompt: \n”);
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

debugger **registers** a **SIGSEGV handler**

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf(“debugger prompt: \n”);
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

at the user-specified breakpoint, the debugger **forces** a SIGSEGV (which its handler will intercept)

Debuggers: advanced breakpoints

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
    - Special register: if PC value = HBP register value, signal

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
    - Special register: if PC value = HBP register value, signal
    - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: "break at instruction X if **some_var = some_value**"

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: “break at instruction X if **some_var = some_value**”
- As before, but signal handler checks if **some_var = some_value**

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: “break at instruction X if **some_var = some_value**”
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: "break at instruction X if **some_var = some_value**"
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt
  - If not, return to program immediately

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: "break at instruction X if **some_var = some_value**"
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt
  - If not, return to program immediately
  - Is this fast or slow?

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: "break at instruction X if **some_var = some_value**"
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt
  - If not, return to program immediately
  - Is this fast or **slow**?

Debuggers: single-stepping

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)
- To implement this, put a breakpoint at the first instruction (= at program start)

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)
- To implement this, put a breakpoint at the first instruction (= at program start)
- The "**single step**" or "**next**" interactive command is equal to:

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)
- To implement this, put a breakpoint at the first instruction (= at program start)
- The "**single step**" or "**next**" interactive command is equal to:
  - Put a breakpoint at the next instruction
  - Resume execution
  - (No, really.)

Debuggers: watchpoints

- You want to know when a variable **changes**

Debuggers: watchpoints

- You want to know when a variable **changes**
- A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Debuggers: watchpoints

- You want to know when a variable **changes**
- A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**
- How could we implement this?

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt
- If not, set next breakpoint and continue (single-step)

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt
- If not, set next breakpoint and continue (single-step)

Hardware Watchpoints:

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt
- If not, set next breakpoint and continue (single-step)

Hardware Watchpoints:

- Special register holds **L**: if the value at address **L** ever changes, the CPU raises an exception

Related tool: profilers

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)
  - Ask the OS to **send you a signal** every X seconds (see `alarm(2)`)

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)
  - Ask the OS to **send you a signal** every X seconds (see `alarm(2)`)
  - In the signal handler you determine the value of the target **program counter** and append it to a growing list file

Related tool: profilers

Definition: A ***profiler*** is a performa
frequency and duration of functio

- **Interpreted languages** provid
  - You **register a function** tha
    program calls a method, lo
    (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)
  - Ask the OS to **send you a signal** every X seconds (see `alarm(2)`)
  - In the signal handler you determine the value of the target **program counter** and append it to a growing list file

This explanation of **sampling** leaves out some things:

- need to map PC values back to procedure names
- need to sum up map results
- sampling is cheap but can miss periodic behavior

Debugging (Part 2/2)

Today's agenda:

- Reading quiz
- Debugging
  - printf debugging and logging
  - debuggers
  - **delta debugging**

Delta debugging: summary

Delta debugging: summary

- ***Delta debugging*** is an automated debugging approach that finds a minimal “interesting” subset of a given set.

Delta debugging: summary

- ***Delta debugging*** is an automated debugging approach that finds a minimal “interesting” subset of a given set.
- Delta debugging is based on **divide-and-conquer** and relies heavily on critical assumptions (**monotonicity**, **unambiguity**, and **consistency**).

Delta debugging: summary

- ***Delta debugging*** is an automated debugging approach that finds a minimal "interesting" subset of a given set.
- Delta debugging is based on **divide-and-conquer** and relies heavily on critical assumptions (**monotonicity**, **unambiguity**, and **consistency**).
- It can be used to find which code changes cause a bug, to minimize failure-inducing inputs, and even to find harmful thread schedules.

Delta debugging: motivation

- Three Problems: One Common Approach
  - Simplifying Failure-Inducing Input
  - Isolating Failure-Inducing Thread Schedules
  - Identifying Failure-Inducing Code Changes

Delta debugging: motivation: inputs

- Having a **test input** may not be enough

Delta debugging: motivation: inputs

- Having a **test input** may not be enough
  - Even if you know the suspicious code, the input may be **too large** to step through

Delta debugging: motivation: inputs

- Having a **test input** may not be enough
  - Even if you know the suspicious code, the input may be **too large** to step through
- This HTML input makes a version of Mozilla crash. Which portion is relevant?

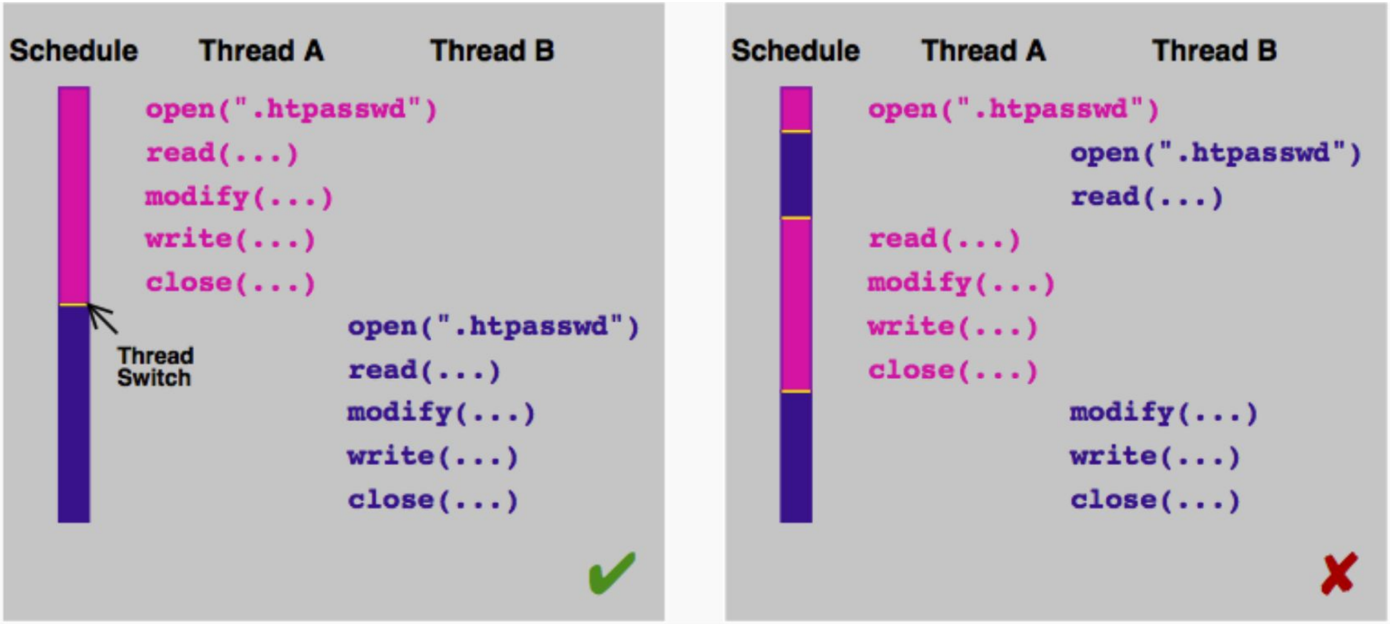
```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
PTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
LUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>

TIPLE SIZE=7>
N VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION

MULTIPLE SIZE=7>
cker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
N VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</table>
```

Implication: delta debugging will be useful for **test input minimization**

Delta debugging: motivation: thread schedules



Delta debugging: motivation: thread schedules

- Multithreaded programs can be **nondeterministic**

Delta debugging: motivation: thread schedules

- Multithreaded programs can be **nondeterministic**
  - Can we find simple, bug-inducing thread schedules?

Delta debugging: motivation: code changes

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)
- 178,000 lines of code have been modified between the two versions

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)
- 178,000 lines of code have been modified between the two versions
  - Where is the bug?
    - … and **which commit** is responsible for introducing it?

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)
- 178,000 lines of code have been modified between the two versions
  - Where is the bug?
    - … and **which commit** is responsible for introducing it?
  - These days: **continuous integration testing** helps
    - … but does not totally solve this. Why?

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream
- Difference in **thread schedule**: difference in the time before a given thread preemption is performed
- Difference in **code**: different statements or expressions in two versions of a program
- Difference in **program state**: different values of internal variables

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream
- Difference in **thread schedule**: difference in the time before a given thread preemption is performed

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream
- Difference in **thread schedule**: difference in the time before a given thread preemption is performed
- Difference in **code**: different statements or expressions in two versions of a program

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream
- Difference in **thread schedule**: difference in the time before a given thread preemption is performed
- Difference in **code**: different statements or expressions in two versions of a program
- Difference in **program state**: different values of internal variables

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - "Find the **smallest subset** of a given set that is still ***interesting***"

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - "Find the **smallest subset** of a given set that is still ***interesting***"
- Abstract solution: **divide-and-conquer**

Delta debugging: unified solution

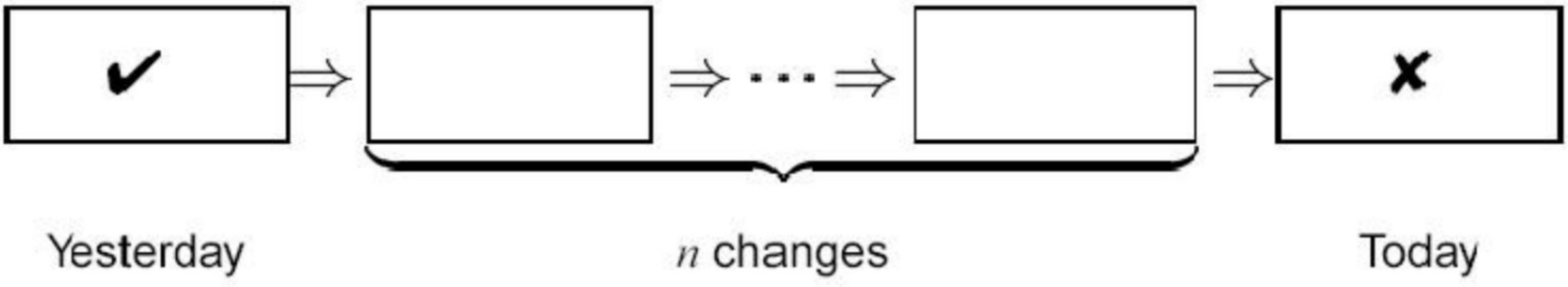
- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - "Find the **smallest subset** of a given set that is still ***interesting***"
- Abstract solution: **divide-and-conquer**
  - **key idea**: split up the set into two subsets, check which of the two is still "**interesting**"

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - "Find the **smallest subset** of a given set that is still ***interesting***"
- Abstract solution: **divide-and-conquer**
  - **key idea**: split up the set into two subsets, check which of the two is still "**interesting**"
  - can be applied to working and failing inputs, code versions, thread schedules, program states, etc.

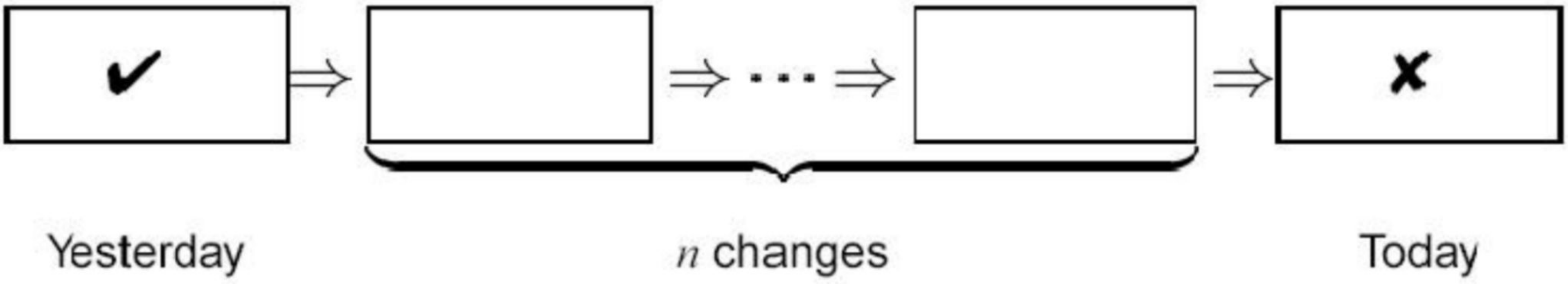
Delta debugging: unified solution

"Yesterday, my program worked. Today, it does not."



Delta debugging: unified solution

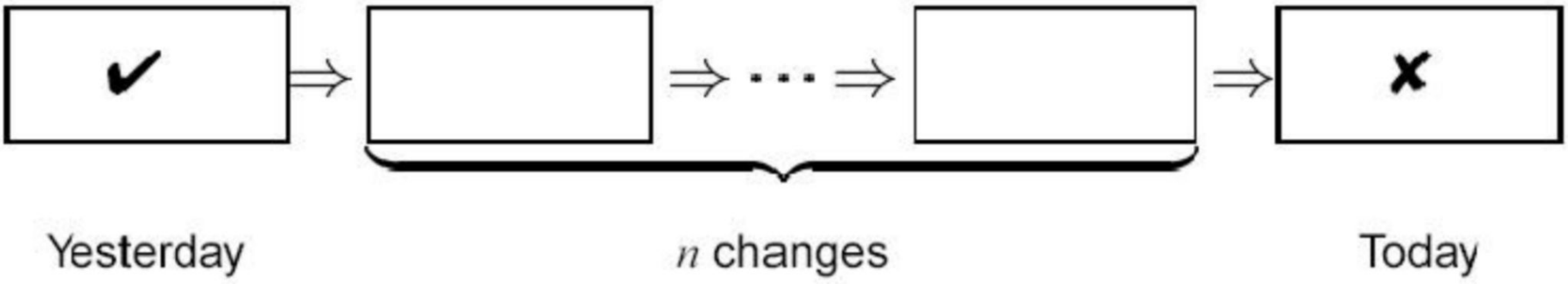
“Yesterday, my program worked. Today, it does not.”



- We will iteratively:

Delta debugging: unified solution

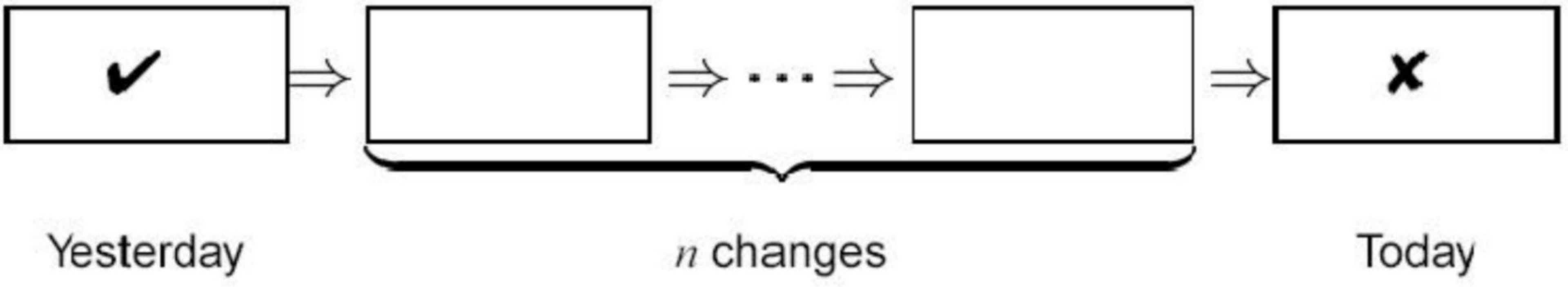
“Yesterday, my program worked. Today, it does not.”



- We will iteratively:
    - **hypothesize** that a small subset is interesting

Delta debugging: unified solution

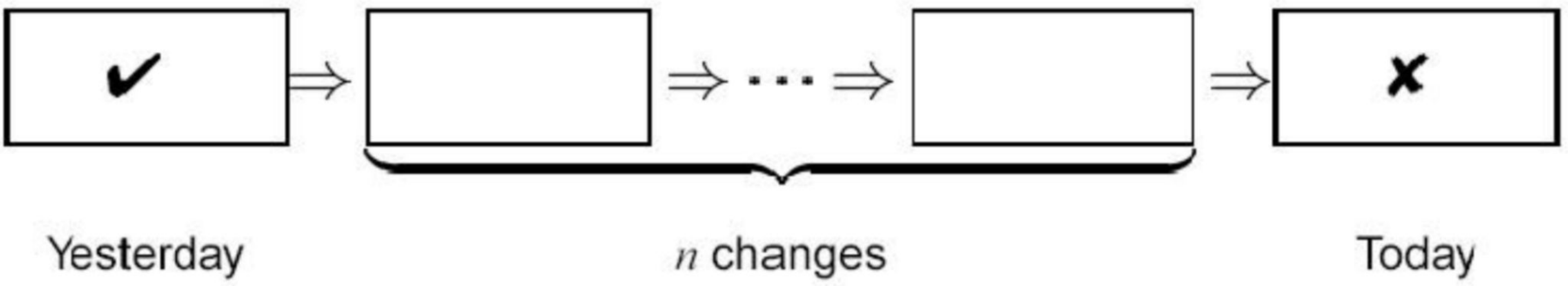
"Yesterday, my program worked. Today, it does not."



- We will iteratively:
  - **hypothesize** that a small subset is interesting
    - e.g., the subset of changes {1, 3, 8} causes the bug

Delta debugging: unified solution

"Yesterday, my program worked. Today, it does not."



- We will iteratively:
  - **hypothesize** that a small subset is interesting
    - e.g., the subset of changes {1, 3, 8} causes the bug
  - run tests to **falsify** our hypothesis

Delta debugging: algorithm

Delta debugging: algorithm

- Given:

Delta debugging: algorithm

- Given:
  - a set **C** = $\{c_1, \ldots, c_n\}$ (of changes)

Delta debugging: algorithm

- Given:
  - a set **C** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C $\rightarrow$ {True, False}

Delta debugging: algorithm

- Given:
  - a set **C** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No

Delta debugging: algorithm

- Given:
  - a set **C** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)

Delta debugging: algorithm

- Given:
  - a set **C** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)
- The **delta debugging algorithm** returns a **minimal Interesting subset** M of C:

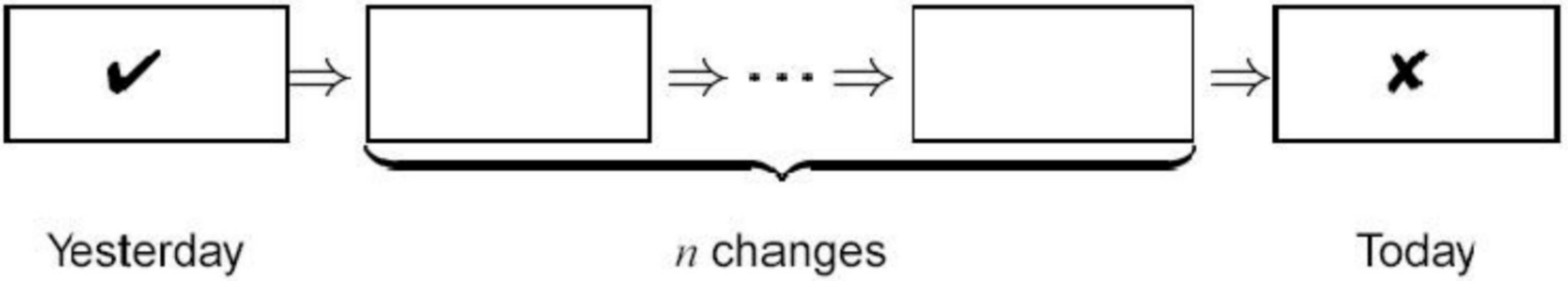
Delta debugging: algorithm

- Given:
  - a set **C** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)
- The **delta debugging algorithm** returns a **minimal Interesting subset** M of C:
  - Interesting(M) = Yes

Delta debugging: algorithm

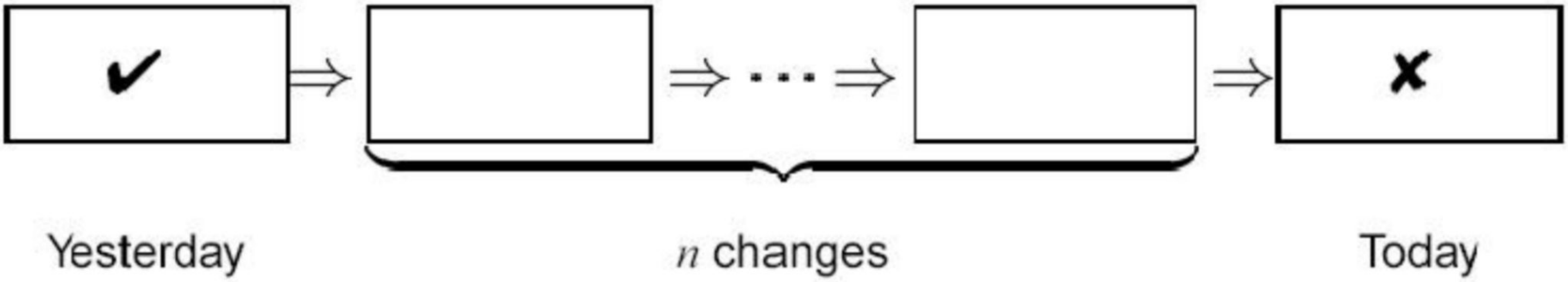
- Given:
  - a set **C** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)
- The **delta debugging algorithm** returns a **minimal Interesting subset** M of C:
  - Interesting(M) = Yes
  - Forall m ⊂ M, Interesting(M - m) = No

Delta debugging: example



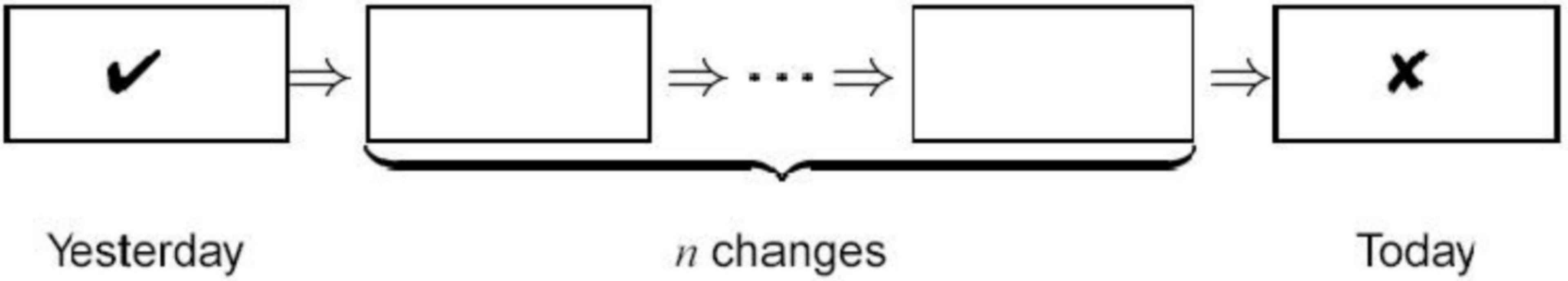
- C =
- Interesting(X) =

Delta debugging: example



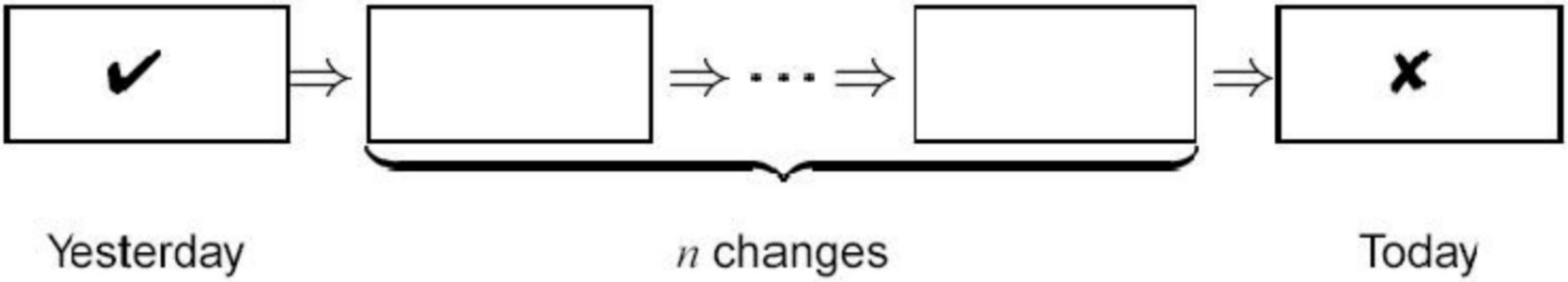
- C = set of *n* changes
- Interesting(X) =

Delta debugging: example



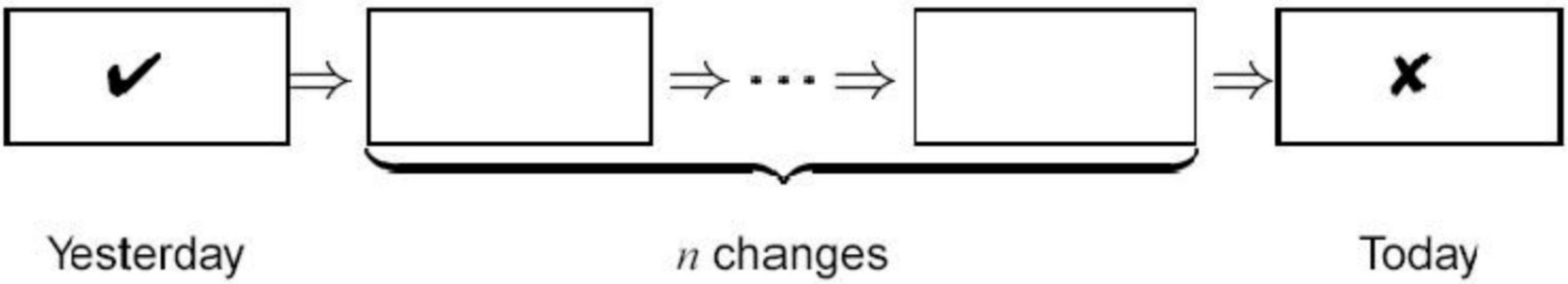
- C = set of *n* changes
- Interesting(X) = apply the changes in in X to Yesterday's version and compile. Run the tests on the result.

Delta debugging: example



- C = set of *n* changes
- Interesting(X) = apply the changes in in X to Yesterday's version and compile. Run the tests on the result.
    - If the tests **fail**, Interesting(X) = **True**.

Delta debugging: example



- C = set of *n* changes
- Interesting(X) = apply the changes in in X to Yesterday's version and compile. Run the tests on the result.
  - If the tests **fail**, Interesting(X) = **True**.
  - If the tests **pass**, Interesting(X) = **False**.

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting
  - **Problem**: if |C| = N, this takes 2^N time

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting
  - **Problem**: if |C| = N, this takes 2^N time
  - Recall: real-world software is **unimaginably huge**

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting
  - **Problem**: if |C| = N, this takes 2^N time
  - Recall: real-world software is **unimaginably huge**
- We want a **polynomial-time** solution
  - Ideally one that is more like log(N)
  - Or we'll loop for what feels like forever

Delta debugging: algorithm candidate

\# Precondition: Interesting({c_1 … c_n }) = True

DD({c, , …, c_n }) =

if n = 1 then return {c_1 }

let P_1 = {c_1 , … $c_{n/2}$}

let P_2 = {$c_{n/2+1}$, …, c_n }

if **Interesting**(P_1) is True:

then return DD(P_1)

else return DD(P_2)

Delta debugging: algorithm candidate

\# Precondition: Interesting($\{c_1 \ldots c_n\}$) = True

DD($\{c, , \ldots, c_n\}$) =

if $n = 1$ then return $\{c_1\}$

let $P_1 = \{c_1, \ldots c_{n/2}\}$

let $P_2 = \{c_{n/2+1}, \ldots, c_n\}$

if **Interesting**(P_1) is True:

then return DD(P_1)

else return DD(P_2)

This is just **binary search**! It won't work if you need a big subset to be Interesting

Delta debugging: algorithm: assumptions

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)
- Interesting is **Monotonic**
  - $\text{Interesting}(X) \rightarrow \text{Interesting}(X \cup \{c\})$

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)
- Interesting is **Monotonic**
  - Interesting(X)$\rightarrow$ Interesting(X $\cup$ {c})
- Interesting is **Unambiguous**
  - Interesting(X) & Interesting(Y) $\rightarrow$ Interesting(X $\cap$ Y)

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)
- Interesting is **Monotonic**
  - Interesting(X)$\rightarrow$ Interesting(X $\cup$ {c})
- Interesting is **Unambiguous**
  - Interesting(X) & Interesting(Y) $\rightarrow$ Interesting(X $\cap$ Y)
- Interesting is **Consistent**
  - Interesting(X) = True xor Interesting(X) = False
  - (Some formulations also allow: Interesting(X) = Unknown)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)

Unambiguous =
Interesting(X) & Interesting(Y) $\rightarrow$
Interesting($X \cap Y$)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True the
  - If Interesting(P_2) = True the
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:

Consistency =
Interesting(X) = True xor
Interesting(X) = False

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:
  - (Interesting(P1) = False) *and* (Interesting(P2) = False)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:
  - (Interesting(P1) = False) *and* (Interesting(P2) = False)
  - What happens in such a case?

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False

Monotonicity =
Interesting(X)$\rightarrow$
Interesting(X $\cup$ {c})

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False
  - Then no subset of P_1 alone or subset of P_2 alone is Interesting

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False
  - Then no subset of P_1 alone or subset of P_2 alone is Interesting
- So the Interesting subset must use a **combination** of elements from P_1 and P_2

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False
  - Then no subset of P_1 alone or subset of P_2 alone is Interesting
- So the Interesting subset must use a **combination** of elements from P_1 and P_2
- In Delta Debugging, this is called ***interference***

Delta debugging: algorithm: interference

- Why is this true?

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True
  - Consider P_2
    - Find a minimal subset D_1 of P_1
    - Such that Interesting($P_2 \cup D_1$) = True

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True
  - Consider P_2
    - Find a minimal subset D_1 of P_1
    - Such that Interesting($P_2 \cup D_1$) = True
  - Then by **Unambiguous**
    - Interesting(($P_1 \cup D_2$) $\cap$ ($P_2 \cup D_1$)) = Interesting($D_1 \cup D_2$) is also minimal

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True
  - Consider P_2
    - Find a minimal subset D_1 of P_1
    - Such that Interesting($P_2 \cup D_1$) = True
  - Then by **Unambiguous**
    - Interesting(($P_1 \cup D_2$) $\cap$ ($P_2 \cup D_1$)) = Interesting($D_1 \cup D_2$) is also minimal

Key point: combination of elements from both

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, ..., 8}
- Let's use DD to find it

1 2 3 4 5 6 7 8 = Interesting

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let’s use DD to find it

1 2 3 4 5 6 7 8 = Interesting

1 2 3 4

5 6 7 8

First step: partition C = {1, …, 8} into P_1 = {1, …, 4} and P_2 = {5, …, 8}

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let’s use DD to find it

1 2 3 4 5 6 7 8	= Interesting
1 2 3 4	= ???
5 6 7 8	= ???

Next step: test P_1 and P_2

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, ..., 8}
- Let's use DD to find it

1 2 3 4 5 6 7 8	= Interesting
1 2 3 4	= False
5 6 7 8	= False

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting(D_1 + P_2)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let’s use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= ???

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting($D_1 + P_2$)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, ..., 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting($D_1 + P_2$)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, ..., 8}
- Let's use DD to find it

```
1 2 3 4 5 6 7 8   = Interesting
1 2 3 4           = False
        5 6 7 8   = False
1 2     5 6 7 8   = False
    3 4 5 6 7 8   = ???
```

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting($D_1 + P_2$)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let’s use DD to find it

1	2	3	4	5	6	7	8	= Interesting
1	2	3	4					= False
				5	6	7	8	= False
1	2			5	6	7	8	= False
		3	4	5	6	7	8	= True
		3		5	6	7	8	= True

D_1 = { 3 }

Now we need to find D_2

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False
3 4	5 6 7 8	= True
3	5 6 7 8	= True
1 2 3 4	5 6	= True

$D_1 = \{ 3 \}$

Now we need to find D_2

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, ..., 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False
3 4	5 6 7 8	= True
3	5 6 7 8	= True
1 2 3 4	5	= False

$D_1 = \{ 3 \}$

Now we need to find D_2

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False
3 4	5 6 7 8	= True
3	5 6 7 8	= True
1 2 3 4	6	= True

$D_1 = \{ 3 \}$

$D_2 = \{ 6 \}$

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False
3 4	5 6 7 8	= True
3	5 6 7 8	= True
1 2 3 4	6	= True

$D_1 = \{ 3 \}$

$D_2 = \{ 6 \}$

So, final answer = $D_1 \cup D_2 = \{ 3, 6 \}$

Delta debugging: final algorithm

\# Precondition: Interesting($\{c_1 \ldots c_n\}$) = True

DD($P, \{c, , \ldots, c_n\}$) =

if $n = 1$ then return $\{c_1\}$

let $P_1 = \{c_1, \ldots c_{n/2}\}$

let $P_2 = \{c_{n/2+1}, \ldots, c_n\}$

if **Interesting**($P_1 \cup P$) is True then return DD(P, P_1)

else if **Interesting**($P_2 \cup P$) is True then return DD(P, P_2)

else return **DD**($P \cup P_2, P_1$) $\cup$ **DD**($P \cup P_1, P_2$)

Delta debugging: algorithmic complexity

Delta debugging: algorithmic complexity

- If a single change induces the failure:
  - DD is **logarithmic**: 2 * log |C|
  - Why?

Delta debugging: algorithmic complexity

- If a single change induces the failure:
  - DD is **logarithmic**: 2 * log |C|
  - Why?
- Otherwise, DD is **linear**
  - Assuming constant time per Interesting() check
  - Is this realistic? ●

Delta debugging: algorithmic complexity

- If a single change induces the failure:
    - DD is **logarithmic**: 2 * log |C|
    - Why?
- Otherwise, DD is **linear**
    - Assuming constant time per Interesting() check
    - Is this realistic? ●
- If Interesting can return “Unknown”
    - DD is **quadratic**: $|C|^2 + 3|C|$
    - If all tests are Unknown except last one (unlikely)

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- **All three** assumptions are **questionable**
- Interesting is **Monotonic**
  - Interesting(X)$\rightarrow$ Interesting($X \cup \{c\}$)
- Interesting is **Unambiguous**
  - Interesting(X) & Interesting(Y) $\rightarrow$ Interesting($X \cap Y$)
- Interesting is **Consistent**
  - Interesting(X) = True xor Interesting(X) = False
  - (Some formulations also allow: Interesting(X) = Unknown)

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- **All three** assumptions are **questionable**
- ~~Interesting is **Monotonic**~~
  - Interesting(X)→ Interesting(X ∪ {c})
- Interesting is **Unambiguous**
  - Interesting(X) & Interes
- Interesting is **Consistent**
  - Interesting(X) = True xo
  - (Some formulations als

Monotonicity is rare in the real world. But DD still finds *an* interesting subset if Interesting is not monotonic (migh tnot be minimal)

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- **All three** assumptions are **questionable**
- Interesting is **Monotonic**
  - Interesting(X)→ Interesting(X ∪ {c})
- ~~Interesting is **Unambiguous**~~
  - Interesting(X) & Interes
- Interesting is **Consistent**
  - Interesting(X) = True xo
  - (Some formulations als

Ambiguity will cause DD to fail. Hint: try tracing DD on Interesting ({2, 8}) = True, but Interesting({2, 8} intersect {3, 6}) = False

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- All three assumptions are
- Interesting is Monotonic
  - Interesting(X)→ Intere
- Interesting is Unambiguou
  - Interesting(X) & Interes
- ~~Interesting is Consistent~~
  - Interesting(X) = True xor Interesting(X) = False
  - (Some formulations also allow: Interesting(X) = Unknown)

The world is **often inconsistent**. Example: we are minimizing changes to a program to find patches that makes it crash. Some subsets may not build or run!

Delta debugging: in the real world

- `git bisect` implements a DD-like algorithm (look it up!)
- for thread schedules: DejaVu tool by IBM, CHESS by Microsoft, etc.
- Eclipse plugins for code changes ("DDinput", "DDchange")
- you can also do delta debugging **by hand** (I do this often for programs that cause compiler bugs!)

Debugging: takeaways

- Debugging is a lot easier when you treat it as a science, rather than an art
- printf debugging and logging are good for determining what causes failures after the fact
- debuggers are fantastic when you want to understand a program's internal state
- delta debugging is a semi-automated approach to formalizing the abstract debugging problem
    - useful way of thinking about how to debug anything
    - try `git bisect`