Technical debt, refactoring, and maintenance (2/2)

Martin Kellogg

Tech debt, refactoring, and maintenance (2/2)

Today's agenda:

- Reading Quiz
- Technical debt: the costs of bad design
- How to pay off technical debt: refactoring

Tech debt, refactoring, and maintenance (2/2)

Today's agenda:

- **Reading Quiz**
- Technical debt: the costs of bad design
- How to pay off technical debt: refactoring

Reading quiz: tech debt 2

Q1: The author describes Netscape making “the single worst strategic mistake that any software company can make”. In **one phrase** (<= 5 words), what mistake did Netscape make?

Q2: The author claims that most programmers, when asked about the system they’re working on, “think the old code is a mess”. He posits this is due to a “fundamental law of programming”. Which one?

A. reading code is harder than writing code
B. the halting problem
C. given enough eyeballs, all bugs are shallow

Reading quiz: tech debt 2

Q1: The author describes Netscape making “the single worst strategic mistake tha ”. In **one phrase** (<= 5 words),

“**rewrite** the code from scratch”

Q2: The author claims that most programmers, when asked about the system they’re working on, “think the old code is a mess”. He posits this is due to a “fundamental law of programming”. Which one?

- **A.** reading code is harder than writing code
- **B.** the halting problem
- **C.** given enough eyeballs, all bugs are shallow

Reading quiz: tech debt 2

Q1: The author describes Netscape making "the single worst strategic mistake tha ". In **one phrase** (<= 5 words),

"**rewrite** the code from scratch"

Q2: The author claims that most programmers, when asked about the system they're working on, "think the old code is a mess". He posits this is due to a "fundamental law of programming". Which one?

A. reading code is harder than writing code
B. the halting problem
C. given enough eyeballs, all bugs are shallow

Tech debt, refactoring, and maintenance (2/2)

Today's agenda:

- Reading Quiz
- **Technical debt: the costs of bad design**
- How to pay off technical debt: refactoring

Review: technical debt

Definition: a ***technical debt*** is a sub-optimal design decision taken intentionally in order to gain some immediate benefit

Review: technical debt

Definition: a ***technical debt*** is a sub-optimal design decision taken intentionally in order to gain some immediate benefit

- Benefits:
  - lower cost (either in dev time or because the code isn't done yet), code reuse, principle of least surprise, avoiding premature optimization, organizational factors, etc.

Review: technical debt

Definition: a ***technical debt*** is a sub-optimal design decision taken intentionally in order to gain some immediate benefit

- Benefits:
  - lower cost (either in dev time or because the code isn't done yet), code reuse, principle of least surprise, avoiding premature optimization, organizational factors, etc.
- Conceptually, when you take on technical debt you are borrowing from **future maintainers** of the system

Review: technical debt

Definition: a ***technical debt*** is a sub-optimal design decision taken intentionally in order to gain some immediate benefit

- Benefits:
  - lower cost (either in dev time or because the code isn't done yet), code reuse, principle of least surprise, avoiding premature optimization, organizational factors, etc.
- Conceptually, when you take on technical debt you are borrowing from **future maintainers** of the system
- a system with technical debt is **harder** to change and reuse

Technical debt: benefits and costs

Examples of debt:

- code smells
- missing tests
- missing documentation
- dependency on old versions of third-party systems
- inefficient and/or non-scalable algorithms

Examples of costs:

- “smelly” code is less flexible
- tests don’t catch breaking change, causing outages
- need to spend time to figure out how to system works
- may need to take over maintenance of old system
- lose potential customers

Technical debt: when is it worth it?

Technical debt: when is it worth it?

- Key consideration:
  - What are the **quality attributes** that our software needs to ultimately satisfy?

Technical debt: when is it worth it?

- Key consideration:
  - What are the **quality attributes** that our software needs to ultimately satisfy?
    - e.g., safety, performance, scalability, etc.

Technical debt: when is it worth it?

- Key consideration:
  - What are the **quality attributes** that our software needs to ultimately satisfy?
    - e.g., safety, performance, scalability, etc.
  - And how do our architectural decisions reflect those attributes?

Technical debt: when is it worth it?

- Key consideration:
  - What are the **quality attributes** that our software needs to ultimately satisfy?
    - e.g., safety, performance, scalability, etc.
  - And how do our architectural decisions reflect those attributes?
    - i.e., will we be able to reach our goals using this design?

Technical debt: when is it worth it?

- Key consideration:
  - What are the **quality attributes** that our software needs to ultimately satisfy?
    - e.g., safety, performance, scalability, etc.
  - And how do our architectural decisions reflect those attributes?
    - i.e., will we be able to reach our goals using this design?
- The choice to take on technical debt is always a **tradeoff**:

Technical debt: when is it worth it?

- Key consideration:
  - What are the **quality attributes** that our software needs to ultimately satisfy?
    - e.g., safety, performance, scalability, etc.
  - And how do our architectural decisions reflect those attributes?
    - i.e., will we be able to reach our goals using this design?
- The choice to take on technical debt is always a **tradeoff**:
  - give up some flexibility later, gain something now

Technical debt: when is it worth it?

- Key consideration:
  - What are the **quality attributes** that our software needs to ultimately satisfy?
    - e.g., safety, performance, scalability, etc.
  - And how do our architectural decisions reflect those attributes?
    - i.e., will we be able to reach our goals using this design?
- The choice to take on technical debt is always a **tradeoff**:
  - give up some flexibility later, gain something now
  - whether this is worthwhile varies **case by case**

Technical debt: when is it worth it?

- Key consideration:
  - What are the **qua** ultimately satisfy
    - e.g., safety, pe
  - And how do our a utes?
    - i.e., will we be able to reach our goals using this design?
- The choice to take on technical debt is always a **tradeoff**:
  - give up some flexibility later, gain something now
  - whether this is worthwhile varies **case by case**

Whether to take on technical debt is often one of the **most consequential** choices you get to make as an engineer. **Take it seriously!**

Technical debt: when is it worth it?

- You should also consider **risk** when taking on technical debt

Technical debt: when is it worth it?

- You should also consider **risk** when taking on technical debt
  - i.e., ask yourself "what is the **worst thing** that could happen in the future if I take this shortcut today"?

Technical debt: when is it worth it?

- You should also consider **risk** when taking on technical debt
  - i.e., ask yourself “what is the **worst thing** that could happen in the future if I take this shortcut today”?
  - risk should preclude you from taking on certain kind of debts
    - e.g., never use laughably-bad security or break laws, even if you don’t plan to deploy this prototype

Technical debt: when is it worth it?

- You should also consider **risk** when taking on technical debt
    - i.e., ask yourself "what is the **worst thing** that could happen in the future if I take this shortcut today"?
    - risk should preclude you from taking on certain kind of debts
        - e.g., never use laughably-bad security or break laws, even if you don't plan to deploy this prototype
- Best practice (especially for relatively risky debts): **write everything down**!
    - that way, you know what you need to fix before releasing

Technical debt: Y2k example

- History quiz: what was the “**Y2k bug**”?

Technical debt: Y2k example

- History quiz: what was the "**Y2k bug**"?
    - Answer: many early programs stored the year using **two digits**
        - assumption: current year = "19" + those two digits

Technical debt: Y2k example

- History quiz: what was the "**Y2k bug**"?
    - Answer: many early programs stored the year using **two digits**
        - assumption: current year = "19" + those two digits
- This is an example of technical debt:

Technical debt: Y2k example

- History quiz: what was the "**Y2k bug**"?
  - Answer: many early programs stored the year using **two digits**
    - assumption: current year = "19" + those two digits
- This is an example of technical debt:
  - **immediate benefit**: saves hard disk space (expensive in 1980)

Technical debt: Y2k example

- History quiz: what was the "**Y2k bug**"?
  - Answer: many early programs stored the year using **two digits**
    - assumption: current year = "19" + those two digits
- This is an example of technical debt:
  - **immediate benefit**: saves hard disk space (expensive in 1980)
  - **long-term cost**: if the program is still being used in 2000, need to fix it!
    - "I just never imagined anyone would be using these systems 10 years later, let alone 20."

[Philippe Kruchten, Robert Nord, Ipek Ozkaya: "Managing Technical Debt: Reducing Friction in Software Development"]

Technical debt: not always strictly technical

- You can also view **other serious risks** to the system's continued maintenance as forms of technical debt

Technical debt: not always strictly technical

- You can also view **other serious risks** to the system's continued maintenance as forms of technical debt
    - e.g., if your ***bus factor*** (= "number of people who need to get hit by a bus before no one understands the system") is low and parts of the system are undocumented…

Technical debt: not always strictly technical

- You can also view **other serious risks** to the system's continued maintenance as forms of technical debt
  - e.g., if your ***bus factor*** (= "number of people who need to get hit by a bus before no one understands the system") is low and parts of the system are undocumented…
    - the amount of technical debt you have is higher than if your bus factor was very high

Technical debt: not always strictly technical

- You can also view **other serious risks** to the system's continued maintenance as forms of technical debt
  - e.g., if your ***bus factor*** (= "number of people who need to get hit by a bus before no one understands the system") is low and parts of the system are undocumented…
    - the amount of technical debt you have is higher than if your bus factor was very high
- Other examples include having **high staff turnover** (which systematically lowers bus factor) or few senior engineers

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a codebase that already exists

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a codebase that already exists
    - we usually call such a codebase ***legacy code***

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a codebase that already exists
  - we usually call such a codebase ***legacy code***
- What if this code **already** has technical debt? (Hint: it **always** does.)

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a codebase that already exists
  - we usually call such a codebase ***legacy code***
- What if this code **already** has technical debt? (Hint: it **always** does.)
  - You **must service** the debt: you must deal with the code as it is

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a codebase that already exists
  - we usually call such a codebase ***legacy code***
- What if this code **already** has technical debt? (Hint: it **always** does.)
  - You **must service** the debt: you must deal with the code as it is
  - You **do not gain** the benefit: the benefit was immediate, but you're reaching the code too late to see it

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a cod
  - we usually
- What if this co ys does.)
  - You **must s** e as it is
  - You **do not** , but you're read

Unfortunate but common anti-pattern:

- dev1 builds a new system, taking on a lot of technical debt
- system is successful initially; dev1 is promoted or moves on
- dev2 is now responsible for paying the debt on the system

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a cod
  - we usually
- What if this co ys does.)
  - You **must s** e as it is
  - You **do not** , but you're reac

Unfortunate but common anti-pattern:

- dev 1 builds a new system, taking on a lot of technical debt
  - system is successful initially, dev 1 is promoted or moves on
  - dev 2 is now responsible for paying the debt on the system :(

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a cod
  - we usually
- What if this co ys does.)
  - You **must s** e as it is
  - You **do not** , but you're read

Unfortunate but common anti-pattern:

- dev 1 builds a new system, taking on a lot of technical debt
- system is successful initially, dev 1 is promoted or moves on
- dev 2 is now responsible for paying the debt on the system :(

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a cod
  - we usually
- What if this co ys does.)
  - You must s e as it is
  - You do not , but
    you're read

Unfortunate but common anti-pattern:

- dev 1 builds a new system, taking on a lot of technical debt
- system is successful initially, dev 1 is promoted or moves on
- dev 2 is now responsible for paying the debt on the system :(

Technical debt: bitrot

Technical debt: bitrot

- Over time, software tends to have **increasing maintenance costs**, even if no technical debt is taken on intentionally

Technical debt: bitrot

- Over time, software tends to have **increasing maintenance costs**, even if no technical debt is taken on intentionally
    - even if the code was initially reviewed and well-designed at the time of commit, and even if changes are reviewed, etc.

Technical debt: bitrot

- Over time, software tends to have **increasing maintenance costs**, even if no technical debt is taken on intentionally
    - even if the code was initially reviewed and well-designed at the time of commit, and even if changes are reviewed, etc.
    - this process is called "***bitrot***"

Technical debt: bitrot

- Over time, software tends to have **increasing maintenance costs**, even if no technical debt is taken on intentionally
  - even if the code was initially reviewed and well-designed at the time of commit, and even if changes are reviewed, etc.
  - this process is called "***bitrot***"
- Why does bitrot happen?

Technical debt: bitrot

- Over time, software tends to have **increasing maintenance costs**, even if no technical debt is taken on intentionally
  - even if the code was initially reviewed and well-designed at the time of commit, and even if changes are reviewed, etc.
  - this process is called "***bitrot***"
- Why does bitrot happen?
  - Systems evolve to meet new needs and add new features

Technical debt: bitrot

- Over time, software tends to have **increasing maintenance costs**, even if no technical debt is taken on intentionally
  - even if the code was initially reviewed and well-designed at the time of commit, and even if changes are reviewed, etc.
  - this process is called "***bitrot***"
- Why does bitrot happen?
  - Systems evolve to meet new needs and add new features
  - Changes happen in dependencies, languages, environment

Technical debt: bitrot

- Over time, software tends to have **increasing maintenance costs**, even if no technical debt is taken on intentionally
  - even if the code was initially reviewed and well-designed at the time of commit, and even if changes are reviewed, etc.
  - this process is called “***bitrot***”
- Why does bitrot happen?
  - Systems evolve to meet new needs and add new features
  - Changes happen in dependencies, languages, environment
  - If the code's structure does not also evolve, it will "rot"

Technical debt example: languages

- **Language choice** is a common example of a place where it might make sense to take on technical debt:

Technical debt example: languages

- **Language choice** is a common example of a place where it might make sense to take on technical debt:
  - relatively-unsafe and/or non-performant languages (e.g., Python, Ruby, JavaScript) are **easier to write** code in

Technical debt example: languages

- **Language choice** is a common example of a place where it might make sense to take on technical debt:
  - relatively-unsafe and/or non-performant languages (e.g., Python, Ruby, JavaScript) are **easier to write** code in
    - but, if you end up needing to write performance-critical or safety-critical code in them, you’re going to have a bad time!

Technical debt example: languages

- **Language choice** is a common example of a place where it might make sense to take on technical debt:
  - relatively-unsafe and/or non-performant languages (e.g., Python, Ruby, JavaScript) are **easier to write** code in
    - but, if you end up needing to write performance-critical or safety-critical code in them, you're going to have a bad time!
  - on the other hand, investing in writing in a safe and performant language (e.g., Rust, Kotlin) has a **higher upfront cost**

Technical debt example: languages

- **Language choice** is a common example of a place where it might make sense to take on technical debt:
  - relatively-unsafe and/or non-performant languages (e.g., Python, Ruby, JavaScript) are **easier to write** code in
    - but, if you end up needing to write performance-critical or safety-critical code in them, you're going to have a bad time!
  - on the other hand, investing in writing in a safe and performant language (e.g., Rust, Kotlin) has a **higher upfront cost**
    - but you might save a big headache later

Technical debt example: languages

- **Language choice** is a common example of a place where it might make sense to take on technical debt:
  - relatively-unsafe and/or non-performant languages (e.g., Python, Ru ... de in
    - but, if y ... ance-critical or safety- ... have a bad time!
  - on the othe ... and performant language (e. ... t cost
    - but you might save a big headache later

Other similar choices include:

- middleware frameworks
- deployment pipeline
- major dependencies

Technical debt example: Facebook + PHP

- Facebook’s original site was written in PHP in 2004

Technical debt example: Facebook + PHP

- Facebook’s original site was written in PHP in 2004
    - PHP is dynamically-typed and **relatively unsafe**

Technical debt example: Facebook + PHP

- Facebook's original site was written in PHP in 2004
    - PHP is dynamically-typed and **relatively unsafe**
        - this caused problems for Facebook as its codebase grew

Technical debt example: Facebook + PHP

- Facebook's original site was written in PHP in 2004
  - PHP is dynamically-typed and **relatively unsafe**
    - this caused problems for Facebook as its codebase grew
- In 2014, Facebook releases **Hack**, a new variant of PHP

Technical debt example: Facebook + PHP

- Facebook's original site was written in PHP in 2004
  - PHP is dynamically-typed and **relatively unsafe**
    - this caused problems for Facebook as its codebase grew
- In 2014, Facebook releases **Hack**, a new variant of PHP
  - Hack added **new safety features** (including gradual typing and type inference)

Technical debt example: Facebook + PHP

- Facebook's original site was written in PHP in 2004
    - PHP is dynamically-typed and **relatively unsafe**
        - this caused problems for Facebook as its codebase grew
- In 2014, Facebook releases **Hack**, a new variant of PHP
    - Hack added **new safety features** (including gradual typing and type inference)
    - "Hack enables us to dynamically convert our code one file at a time" - Facebook Technical Lead, HipHop VM (HHVM)

Paying down technical debt

- It is possible to **reduce** the amount of technical debt in a codebase by improving its design

Paying down technical debt

- It is possible to **reduce** the amount of technical debt in a codebase by improving its design
  - one option: **rewriting** the whole system (but think about next class' reading!)

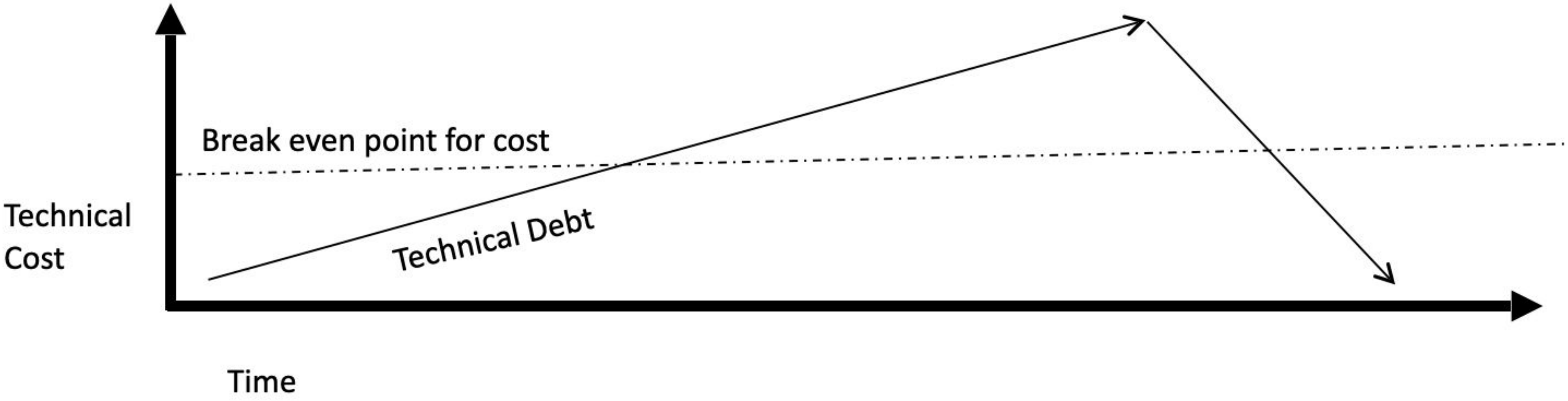
Paying down technical debt

- It is possible to reduce the amount of technical debt in a codebase by improving its design
  - one option: **rewriting** the whole system (but think about next class' reading!)
  - more common: **refactoring** the code

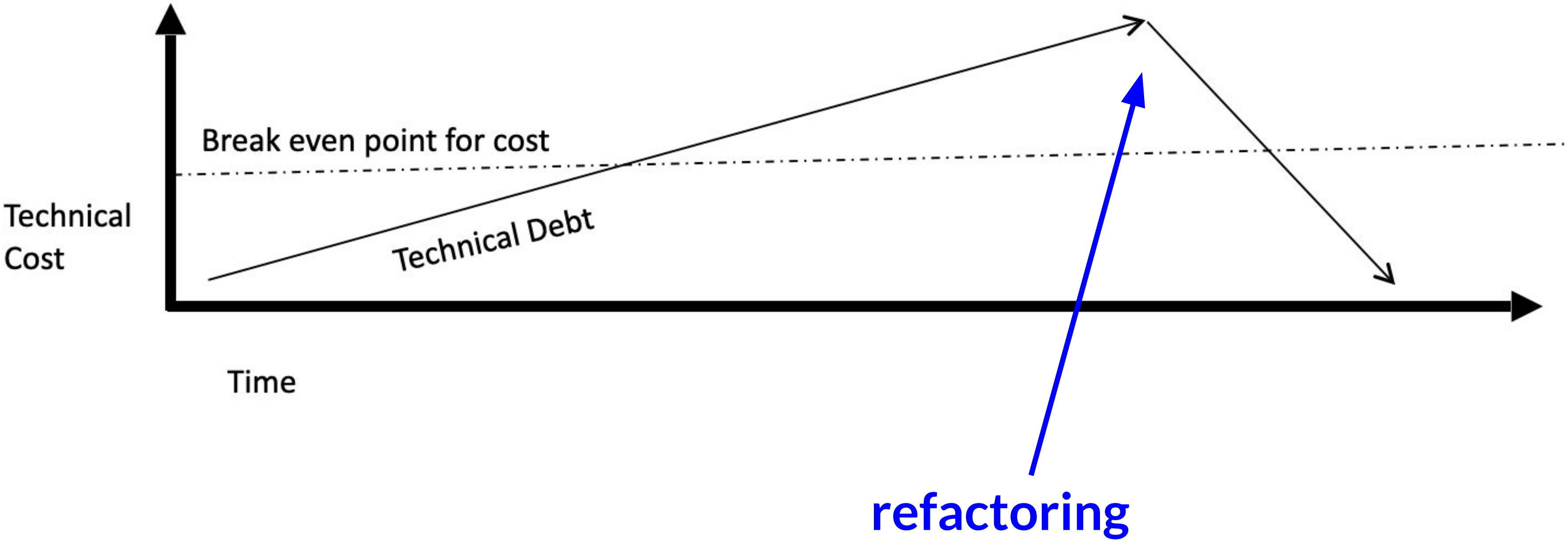
Paying down technical debt

- It is possible to **reduce** the amount of technical debt in a codebase by improving its design
  - one option: **rewriting** the whole system (but think about next class' reading!)
  - more common: **refactoring** the code
- ***refactoring*** is the process of applying behaviour-preserving transformations (called ***refactorings***) to a program, with the goal of improving its non-functional properties (e.g., design, performance)

Paying down technical debt



Paying down technical debt



Paying down technical debt: best practices

Paying down technical debt: best practices

- Advice: set aside **specific time** to pay off technical debt

Paying down technical debt: best practices

- Advice: set aside **specific time** to pay off technical debt
    - Google has (had?) “20% time” for tasks like this

Paying down technical debt: best practices

- Advice: set aside **specific time** to pay off technical debt
    - Google has (had?) “20% time” for tasks like this
- **New projects** can take on some technical debt

Paying down technical debt: best practices

- Advice: set aside **specific time** to pay off technical debt
  - Google has (had?) “20% time” for tasks like this
- **New projects** can take on some technical debt
  - i.e., refactoring at the start of a project to make the rest of the new code easier to write

Paying down technical debt: best practices

- Advice: set aside **specific time** to pay off technical debt
  - Google has (had?) “20% time” for tasks like this
- **New projects** can take on some technical debt
  - i.e., refactoring at the start of a project to make the rest of the new code easier to write
- Have a plan: **don’t put off dealing with technical debt indefinitely**

Paying down technical debt: best practices

- Advice: set aside **specific time** to pay off technical debt
  - Google has (had?) “20% time” for tasks like this
- **New projects** can take on some technical debt
  - i.e., refactoring at the start of a project to make the rest of the new code easier to write
- Have a plan: **don’t put off dealing with technical debt indefinitely**
  - When a crisis hits, it’s too late
  - Hasty fixes to unmaintainable code likely to multiply problems!
  - Eventually, mounting technical debt can bury a team

Tech debt, refactoring, and maintenance (1/2)

Today's agenda:

- Finish design pattern slides
- Reading Quiz
- Technical debt: the costs of bad design
- **How to pay off technical debt: refactoring**

Refactoring

Definition: ***refactoring*** is improving a piece of software's internal structure without altering its external behavior.

Refactoring

Definition: ***refactoring*** is improving a piece of software's internal structure without altering its external behavior.

- Incurs a short-term time/work cost to reap **long-term benefits**

Refactoring

Definition: ***refactoring*** is improving a piece of software's internal structure without altering its external behavior.

- Incurs a short-term time/work cost to reap **long-term benefits**
- A long-term **investment** in the overall quality of your system.

Refactoring

Definition: ***refactoring*** is improving a piece of software's internal structure without altering its external behavior.

- Incurs a short-term time/work cost to reap **long-term benefits**
- A long-term **investment** in the overall quality of your system.

What refactoring is **not**:

Refactoring

Definition: ***refactoring*** is improving a piece of software's internal structure without altering its external behavior.

- Incurs a short-term time/work cost to reap **long-term benefits**
- A long-term **investment** in the overall quality of your system.

What refactoring is **not**:

- rewriting code
- adding features
- debugging code

Aside: rewriting code

- “refactoring code” != “rewriting code”

Aside: rewriting code

- “refactoring code” != “rewriting code”
- **key difference**: when you refactor code, you are modifying the old version (and keeping all of its **accumulated bug fixes**, etc.)

Aside: rewriting code

- “refactoring code” != “rewriting code”
- **key difference**: when you refactor code, you are modifying the old version (and keeping all of its **accumulated bug fixes**, etc.)
  - if you rewrite from scratch, you might end up with a **worse system** than you started with!

Aside: rewriting code

- “refactoring code” != “rewriting code”
- **key difference**: when you refactor code, you are modifying the old version (and keeping all of its **accumulated bug fixes**, etc.)
    - if you rewrite from scratch, you might end up with a **worse system** than you started with!
- rewriting *is* sometimes worthwhile or necessary

Aside: rewriting code

- “refactoring code” != “rewriting code”
- **key difference**: when you refactor code, you are modifying the old version (and keeping all of its **accumulated bug fixes**, etc.)
  - if you rewrite from scratch, you might end up with a **worse system** than you started with!
- rewriting *is* sometimes worthwhile or necessary
  - fundamentally incompatible with new requirements
  - “build one to throw away” (i.e., prototyping)
  - old Google promotion system

Aside: rewriting code

- "refactoring code" !=
- **key difference**: when
  old version (and keep
  - if you rewrite fro
    system than you started with!
- rewriting *is* sometimes worthwhile or necessary
  - fundamentally incompatible with new requirements
  - "build one to throw away" (i.e., prototyping)
  - old Google promotion system

Advice:

- even if rewriting is necessary, don't totally abandon the old system
- keep old tests/CI jobs, and don't release the new system until they pass

Refactoring: motivation

Question: why fix a part of your system that **isn't broken**?

Refactoring: motivation

Question: why fix a part of your system that **isn't broken**?

- Each part of your system's code has three purposes:
  - to execute its functionality,
  - to allow change,
  - to communicate well to developers who read it.

Refactoring: motivation

Question: why fix a part of your system that **isn't broken**?

- Each part of your system's code has three purposes:
  - to execute its functionality,
  - to allow change,
  - to communicate well to developers who read it.
- If the code does not do one or more of these, it ***is*** broken.

Refactoring: motivation

Question: why fix a part of your system that **isn't broken**?

- Each part of your system's code has three purposes:
  - to execute its functionality,
  - to allow change,
  - to communicate well to developers who read it.
- If the code does not do one or more of these, it ***is*** broken.
- Refactoring should improve the software's design:
  - more extensible, flexible, understandable, performant, …
  - every design improvement has costs (and risks)

Refactoring: when to refactor

Refactoring: when to refactor

Definition: a “*code smell*” is a minor design issue with a piece of code that is not a defect *per se*, but is still undesirable

Refactoring: when to refactor

Definition: a "***code smell***" is a minor design issue with a piece of code that is not a defect *per se*, but is still undesirable

- intuition: each code smell is an **irritation** on its own, but in large groups they impede maintenance

Refactoring: when to refactor

Definition: a "***code smell***" is a minor design issue with a piece of code that is not a defect *per se*, but is still undesirable

- intuition: each code smell is an **irritation** on its own, but in large groups they impede maintenance
- many code smells -> good idea to refactor

Refactoring: when to refactor

Definition: a "***code smell***" is a minor design issue with a piece of code that is not a defect *per se*, but is still undesirable

- intuition: each code smell is an **irritation** on its own, but in large groups they impede maintenance
- many code smells -> good idea to refactor
- a good refactoring often fixes more than one code smell
  - sometimes many more than one

Refactoring: when to refactor

Examples of **common code smells**:

Refactoring: when to refactor

Examples of **common code smells**:

- Duplicated code
- Poor abstraction (change one place → must change others)
- Large loop, method, class, parameter list; deeply nested loop
- Module has too little cohesion
- Modules have too much coupling
- Module has poor encapsulation
- Dead code
- Design is unnecessarily general
- Design is too specific

Refactoring: “low-level” refactoring

- “*low-level*” refactorings are small changes to the code that mitigate or remove one or more code smells. Examples:

Refactoring: “low-level” refactoring

- “***low-level***” refactorings are small changes to the code that mitigate or remove one or more code smells. Examples:
  - Renaming (methods, variables)
  - Naming (extracting) “magic” constants
  - Extracting common functionality (including duplicate code) into a module/method/etc.
  - Changing method signatures
  - Splitting one method into two or more to improve cohesion and readability (by reducing its size)

also see https://refactoring.com/catalog/

Refactoring: “low-level” refactoring

- modern IDEs have good support for low-level refactoring

Refactoring: "low-level" refactoring

- modern IDEs have good support for low-level refactoring
    - ***IDE*** = "***integrated development environment***"
        - e.g., Eclipse, VSCode, IntelliJ, etc.

Refactoring: “low-level” refactoring

- modern IDEs have good support for low-level refactoring
  - ***IDE*** = “***integrated development environment***”
    - e.g., Eclipse, VSCode, IntelliJ, etc.
- they automate:
  - renaming of variables, methods, classes
  - extraction of methods and constants
  - extraction of repetitive code snippets
  - changing method signatures
  - warnings about inconsistent code
  - …

Refactoring: “high-level” refactoring

Refactoring: “high-level” refactoring

- “***High-level***” refactoring might include:

Refactoring: “high-level” refactoring

- “*High-level*” refactoring might include:
  - Refactoring to design patterns
  - Changing language idioms (safety, brevity)
  - Performance optimization
  - Clarifying a statement that has evolved over time or is unclear

Refactoring: “high-level” refactoring

- “***High-level***” refactoring might include:
  - Refactoring to design patterns
  - Changing language idioms (safety, brevity)
  - Performance optimization
  - Clarifying a statement that has evolved over time or is unclear
- Compared to low-level refactoring, high-level is:

Refactoring: "high-level" refactoring

- "***High-level***" refactoring might include:
    - Refactoring to design patterns
    - Changing language idioms (safety, brevity)
    - Performance optimization
    - Clarifying a statement that has evolved over time or is unclear
- Compared to low-level refactoring, high-level is:
    - Not as well-supported by tools
    - But much **more important**!

Refactoring: how to refactor

- When you identify an area of your system that:

Refactoring: how to refactor

- When you identify an area of your system that:
  - is **poorly designed**, and

Refactoring: how to refactor

- When you identify an area of your system that:
    - is **poorly designed**, and
    - is **poorly tested** (even if it seems to work so far), and

Refactoring: how to refactor

- When you identify an area of your system that:
  - is **poorly designed**, and
  - is **poorly tested** (even if it seems to work so far), and
  - now **needs new features**...

Refactoring: how to refactor

- When you identify an area of your system that:
  - is **poorly designed**, and
  - is **poorly tested** (even if it seems to work so far), and
  - now **needs new features**…

These are a good set of criteria for deciding to refactor code

- especially “needs new features”, because if you don’t refactor you’ll be **paying interest** on the tech debt!

Refactoring: how to refactor

- When you identify an area of your system that:
    - is **poorly designed**, and
    - is **poorly tested** (even if it seems to work so far), and
    - now **needs new features**…
- What should you do?

Refactoring: how to refactor

- When you identify an area of your system that:
  - is **poorly designed**, and
  - is **poorly tested** (even if it seems to work so far), and
  - now **needs new features**...
- What should you do?
  - Write **unit tests** that verify the code's external correctness. (They should pass on the current, badly-designed code.)

Refactoring: how to refactor

- When you identify an area of your system that:
    - is **poorly designed**, and
    - is **poorly tested** (even if it seems to work so far), and
    - now **needs new features**...
- What should you do?
    - Write **unit tests** that verify the code's external correctness. (They should pass on the current, badly-designed code.)
    - **Refactor** the code. (Some unit tests may break. Fix the bugs.)

Refactoring: how to refactor

- When you identify an area of your system that:
  - is **poorly designed**, and
  - is **poorly tested** (even if it seems to work so far), and
  - now **needs new features**...
- What should you do?
  - Write **unit tests** that verify the code's external correctness. (They should pass on the current, badly-designed code.)
  - **Refactor** the code. (Some unit tests may break. Fix the bugs.)
  - Add any **new features**.

Refactoring: how to refactor

- When you identify an area of your system that:
  - is **poorly designed**, and
  - is **poorly tested** (even if it seems to work so far), and
  - now **needs new features**...
- What should you do?
  - Write **unit tests** that verify the code's external correctness. (They should pass on the current, badly-designed code.)
  - **Refactor** the code. (Some unit tests may break. Fix the bugs.)
  - Add any **new features**.
  - As always, keep changes small, do code reviews, etc.

Tech debt & refactoring: takeaways

- most real systems have some amount of technical debt
- taking on technical debt can be an effective way to meet goals, but it also comes with significant costs. Consider the choice to take on tech debt carefully.
- refactoring is the best method to “pay down” tech debt
- when refactoring, be sure to maintain the current behaviors of the system: refactorings should be functionally-identical
- avoid rewriting a whole system unless you absolutely have to
    - prefer to gradually refactor a “bad” system over time
- set aside time in your schedule to pay down tech debt