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

Martin Kellogg

Reading quiz: technical debt

Q1: Which of these did Spolsky use as an example of “ugly” code?

A. a method called “fixUp” that undid the effect of another method
B. a datatype called “FuckedString”
C. a six-dimensional array
D. Hungarian notation

Q2: **TRUE** or **FALSE**: “Friday Deploy Freezes Are Exactly Like Murdering Puppies” because the author is advocating for a “996” working week, so that their startup can compete with their Chinese competitors

Reading quiz: technical debt

Q1: Which of these did Spolsky use as an example of “ugly” code?

A. a method called “fixUp” that undid the effect of another method
B. a datatype called “FuckedString”
C. a six-dimensional array
D. Hungarian notation

Q2: **TRUE** or **FALSE**: “Friday Deploy Freezes Are Exactly Like Murdering Puppies” because the author is advocating for a “996” working week, so that their startup can compete with their Chinese competitors

Reading quiz: technical debt

Q1: Which of these did Spolsky use as an example of “ugly” code?

A. a method called “fixUp” that undid the effect of another method

B. a datatype called “FuckedString”

C. a six-dimensional array

D. Hungarian notation

Q2: **TRUE** or **FALSE**: “Friday Deploy Freezes Are Exactly Like Murdering Puppies” because the author is advocating for a “996” working week, so that their startup can compete with their Chinese competitors

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: 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 strictly technical

- Consider the example from the reading: “***Friday Deploy Freezes***”

Technical debt: not always strictly technical

- Consider the example from the reading: "***Friday Deploy Freezes***"
  - idea: to avoid an outage over the weekend that will cause the team to get paged, don't deploy new code on Fridays

Technical debt: not always strictly technical

- Consider the example from the reading: "***Friday Deploy Freezes***"
  - idea: to avoid an outage over the weekend that will cause the team to get paged, don't deploy new code on Fridays
- Why is this an example of technical debt?

Technical debt: not always strictly technical

- Consider the example from the reading: "***Friday Deploy Freezes***"
  - idea: to avoid an outage over the weekend that will cause the team to get paged, don't deploy new code on Fridays
- Why is this an example of technical debt?
  - **Short-term benefit**: can enjoy time with family, brunch, etc.
  - **Long-term cost**: new code gets to users slower, tougher to debug problems if they occur, etc.

Technical debt: not always strictly technical

- Consider the example from the reading: "***Friday Deploy Freezes***"
  - idea: to avoid an outage over the weekend that will cause the team to get paged, don't deploy new code on Fridays
- Why is this an example of technical debt?
  - **Short-term benefit**: can enjoy time with family, brunch, etc.
  - **Long-term cost**: new code gets to users slower, tougher to debug problems if they occur, etc.
- Author: "**Anxiety related to deploys is the single largest source of technical debt** in many, many orgs"

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 reac

Unfortunate but common anti-pattern:

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

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a cod
  - we usually
- What if this co
  - You **must s**
  - You **do not**
    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

ys does.)

e as it is

, but

Technical debt: not always your fault

- Common situation: you are now responsible for maintaining and improving a cod
  - we usually
- What if this co
  - You **must s**
  - You **do not**

ys does.)
e as it is
, but

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.g it 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)

Technical debt example: machine learning

Technical debt example: machine learning

- Machine-learning components can encourage tech debt

Technical debt example: machine learning

- Machine-learning components can encourage tech debt
  - hard to enforce **strict abstraction boundaries**

Technical debt example: machine learning

- Machine-learning components can encourage tech debt
  - hard to enforce **strict abstraction boundaries**
  - after all, one big reason for ML is that the desired behavior **cannot be effectively implemented in software logic** without dependency on external data!

Technical debt example: machine learning

- Machine-learning components can encourage tech debt
  - hard to enforce **strict abstraction boundaries**
  - after all, one big reason for ML is that the desired behavior **cannot be effectively implemented in software logic** without dependency on external data!
- For this reason, Google engineers have called ML systems the "**high-interest credit card**" of technical debt [1]

[1] Sculley, David, et al. *"Machine learning: The high interest credit card of technical debt."* SE4ML: software engineering for machine learning (NIPS 2014 Workshop)

Technical debt example: machine learning

- Machine-learning components can encourage tech debt
  - hard to enforce **strict abstraction boundaries**
  - after all, one big reason for ML is that the desired behavior **cannot be effectively implemented in software logic** without dependency on external data!
- For this reason, Google engineers have called ML systems the "**high-interest credit card**" of technical debt [1]
  - can get you a lot of value in the short term!

[1] Sculley, David, et al. *"Machine learning: The high interest credit card of technical debt."* SE4ML: software engineering for machine learning (NIPS 2014 Workshop)

Technical debt example: machine learning

- Machine-learning components can encourage tech debt
  - hard to enforce **strict abstraction boundaries**
  - after all, one big reason for ML is that the desired behavior **cannot be effectively implemented in software logic** without dependency on external data!
- For this reason, Google engineers have called ML systems the "**high-interest credit card**" of technical debt [1]
  - can get you a lot of value in the short term!
  - but if you don't pay down the debt quickly…

[1] Sculley, David, et al. *"Machine learning: The high interest credit card of technical debt."* SE4ML: software engineering for machine learning (NIPS 2014 Workshop)

Aside: LLMs and technical debt

Aside: LLMs and technical debt

- It is **not yet well understood** how LLM code generators (e.g., Calude Code and friends) interact with technical debt

Aside: LLMs and technical debt

- It is **not yet well understood** how LLM code generators (e.g., Calude Code and friends) interact with technical debt
- However, early signs are **not promising**:

Aside: LLMs and technical debt

- It is **not yet well understood** how LLM code generators (e.g., Calude Code and friends) interact with technical debt
- However, early signs are **not promising**:
  - LLMs seem to be easily confused by atypical code patterns, quirks of leaky abstractions, etc. (all hallmarks of tech debt)

Aside: LLMs and technical debt

- It is **not yet well understood** how LLM code generators (e.g., Calude Code and friends) interact with technical debt
- However, early signs are **not promising**:
  - LLMs seem to be easily confused by atypical code patterns, quirks of leaky abstractions, etc. (all hallmarks of tech debt)
  - LLMs can **introduce technical debt** themselves
    - e.g., studies have shown that with an LLM assistant, devs are more likely to write insecure code [1]

[1] *Do Users Write More Insecure Code with AI Assistants?* Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. CCS 2023.

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 today's Spolsky reading!)

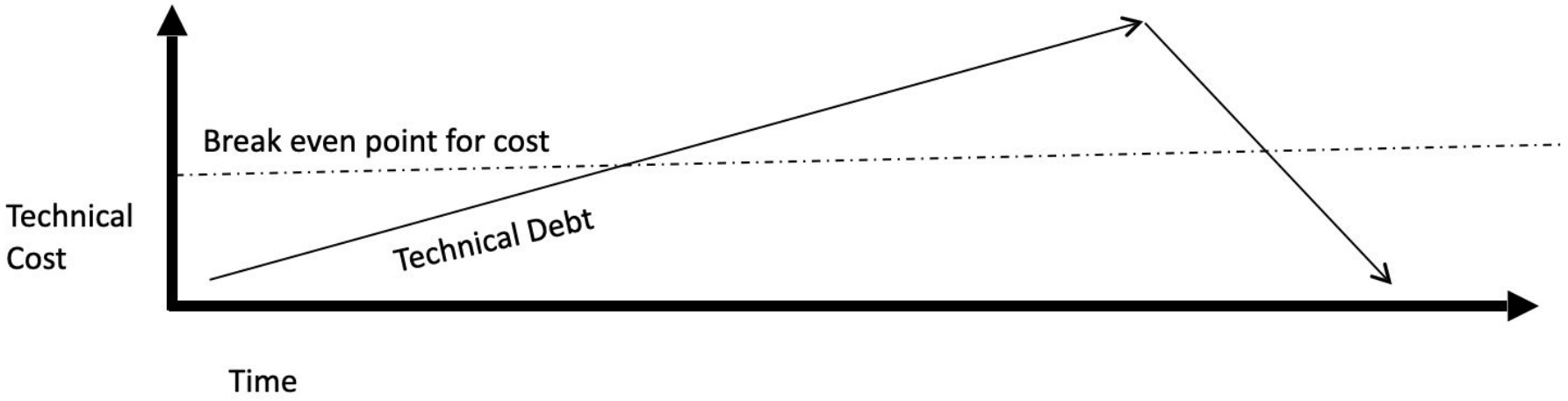
Paying down technical debt

- 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 today's Spolsky reading!)
  - more common: **refactoring** the code

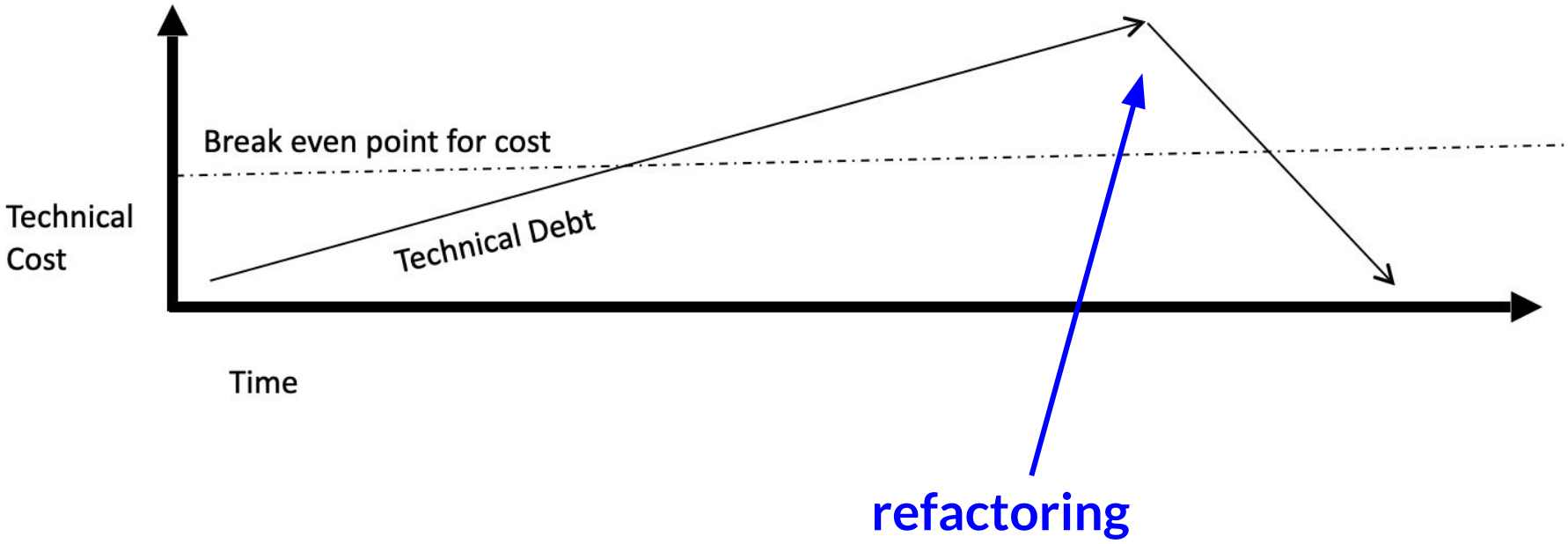
Paying down technical debt

- 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 today's Spolsky 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

Agenda:

- Finish design pattern slides
- 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

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: “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, me
  - extraction of methods an
  - extraction of repetitive c
  - changing method signatu
  - warnings about inconsistent code
  - …

My advice/opinion: don’t rely on your IDE too much. It’s useful for auto-complete, simple refactoring, red squiggles, etc. But, if you let it control the build process you’ll have a bad time.

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
  - is **poorly designed**, an
  - is **poorly tested** (even
  - now **needs new featu**
- 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.)

Q: This process is an example of what kind of testing that we discussed earlier in this class?

Refactoring: how to refactor

- When you identify an area
  - is **poorly designed**, an
  - is **poorly tested** (even
  - now **needs new featu**
- 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.)

Q: This process is an example of what kind of testing that we discussed earlier in this class?

A: **differential** testing

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.

Takeaways: tech debt and refactoring

- Technical debt accrues when you take a shortcut for some immediate benefit that makes a system harder to maintain
  - tech debt is inevitable in large systems
  - but you should be thoughtful about when/how you take it on!
- When and how you take on technical debt is one of the biggest judgment calls that you will make as a low-level engineer
- Refactoring is the process of improving a codebase's non-functional properties while maintaining its behavior
  - refactoring is a useful way to reduce tech debt
  - you often want to pair refactoring with adding new features