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

Martin Kellogg

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

Today's agenda:

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

Software Architecture (Part 2 of 2)

Agenda:

- Strategies for good design
- Design patterns
  - Structural patterns
  - **Creational patterns**
  - Behavioural patterns

Design patterns: creational patterns

Design patterns: creational patterns

- ***Creational design patterns*** avoid complexity by controlling object creation so that objects are created in a manner suitable for the situation. They make a system **independent of how its objects are created**.

Design patterns: creational patterns

- ***Creational design patterns*** avoid complexity by controlling object creation so that objects are created in a manner suitable for the situation. They make a system **independent of how its objects are created**.
- A plain constructor **may not allow** you to:

Design patterns: creational patterns

- ***Creational design patterns*** avoid complexity by controlling object creation so that objects are created in a manner suitable for the situation. They make a system **independent of how its objects are created**.
- A plain constructor **may not allow** you to:
    - Control how and when an object is used
    - Overcome language limitations (e.g., no default arguments)
    - Hide polymorphic types
    - Specify different combinations of optional arguments

Design patterns: creational patterns

- ***Creational design patterns*** avoid complexity by controlling object creation so that objects ar … e situation. They make a sys… **are created**.
- A plain constructor **may not allow** you to:
  - Control how and when an object is used
  - Overcome language limitations (e.g., no default arguments)
  - Hide polymorphic types
  - Specify different combinations of optional arguments

> Different creational patterns allow you to overcome these limitations of simple constructors

Creational patterns: named constructor

- In the ***Named Constructor Pattern***, you declare the class's normal constructors to be private or protected and make a public static creation method.

Creational patterns: named constructor

- In the ***Named Constructor Pattern***, you declare the class's normal constructors to be private or protected and make a public static creation method.

```
class Llama {
public:
 static Llama* create_llama(string name) {
 return new Llama(name);
 }
private: // Making ctor private
 Llama(string name_in): name(name_in) {}
 string name;
};
```

Creational patterns: named constructor

- In the *Named Constructor Patter* constructors to be private or pr creation method.

```
class Llama {
public:
 static Llama* create_llama(string n
 return new Llama(name);
 }
private: // Making ctor private
 Llama(string name_in): name(name_in) {}
 string name;
};
```

Why might you do this?

- might want to change to Llama subclass later
- want to validate arguments from clients, but make construction fast internally
- etc.

Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client

Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client
  - Recall: design for maintainability and extensibility. We don't want the client to depend on (and thus “lock in”) the actual subtypes.

Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client
  - Recall: design for maintainability and extensibility. We don't want the client to depend on (and thus “lock in”) the actual subtypes.
- The typical solution is to write a function that creates objects of the type we want but returns that object so that it appears to be (“cast to”) a member of the base class

Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client
  - Recall: design for maintainability and extensibility. We don't want the client to depend on (and thus "lock in") the actual subtypes.
- The typical solution is to write a function that creates objects of the type we want but returns that object so that it appears to be ("cast to") a member of the base class
  - this is a specific variant of the named constructor pattern

Creational patterns: factories

- The ***factory method pattern*** (or just ***factory pattern***) is a creational design pattern that uses factory methods to create objects without having the return type reveal the exact subclass created.

Creational patterns: factories

- The ***factory method pattern*** (or just ***factory pattern***) is a creational design pattern that uses factory methods to create objects without having the return type reveal the exact subclass created.

```
Payment * payment_factory(string name, string type) {
 if (type == "credit_card")
   return new CreditCardPayment(name);
 else if (type == "bitcoin")
   return new BitcoinPayment(name);
 … }

Payment * webapp_session_payment =
 payment_factory(customer_name, "credit_card");
```

Creational patterns: factories

- The *factory method pattern* (or
  design pattern that uses facto
  without having the return type

Note how the implementation details are hidden from the client, and they can only treat the result as a generic payment

```
Payment * payment_factory(string name, string type) {
 if (type == "credit_card")
   return new CreditCardPayment(name);
 else if (type == "bitcoin")
   return new BitcoinPayment(name);
 … }

Payment * webapp_session_payment =
 payment_factory(customer_name, "credit_card");
```

Creational patterns: factories

- You may also encounter implementations in which special methods create the right type:

Creational patterns: factories

- You may also encounter implementations in which special methods create the right type:

```
class PaymentFactory {
public:
 static Payment* make_credit_payment(string name){
   return new CreditCardPayment(name);
 }
 static Payment* make_bc_payment(string name){
   return new BitcoinPayment(name);
 }};
Payment * webapp_session_payment =
PaymentFactory::make_credit_payment(customer_name);
```

Creational patterns: example

- Suppose we're implementing a computer game with a **polymorphic Enemy class hierarchy**, and we want to spawn **different versions** of enemies based on the difficulty level.
- e.g., normal difficulty = regular Goomba

- hard difficulty = spiked Goomba

Creational patterns: example: anti-patterns

- An ***anti-pattern*** is a common response to a recurring problem that is usually ineffective and risks being counterproductive.

Creational patterns: example: anti-patterns

- An ***anti-pattern*** is a common response to a recurring problem that is usually ineffective and risks being counterproductive.
- A bad solution (i.e., anti-pattern) would be to check the difficulty at each of the many places in the code related to spawning enemies:

```
Enemy* goomba = nullptr;
if (difficulty == “normal”)
  goomba = new Goomba();
else if (difficulty == “hard”)
  goomba = new SpikedGoomba();
```

Creational patterns: example: anti-patterns

- An ***anti-pattern*** is a common response to a recurring problem that is usually ineffective and risks being counterproductive.
- A bad solution (i.e., anti-pattern) would be to check the difficulty at each of the many places in the code related to spawning enemies:

```
Enemy* goomba = nullptr;
if (difficulty == “normal”)
  goomba = new Goomba();
else if (difficulty == “hard”)
  goomba = new SpikedGoomba();
```

Why is this bad?

- code duplication
- consider how you'd add a new difficulty level...

Creational patterns: example: anti-patterns

- An ***anti-pattern*** is a common response to a recurring problem that is usually ineffective and risks being counterproductive.
- A bad solution (i.e., anti-pattern) would be to check the difficulty at each of the many places in the code related to spawning enemies:

```
Enemy* goomba = nullptr;
if (difficulty == "normal")
  goomba = new Goomba();
else if (difficulty == "hard")
  goomba = new SpikedGoomba();
```

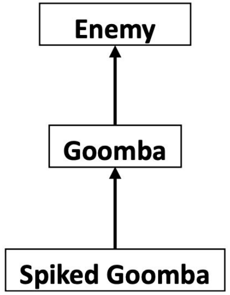
Why is this bad?

- code duplication
- consider how you'd add a new difficulty level...

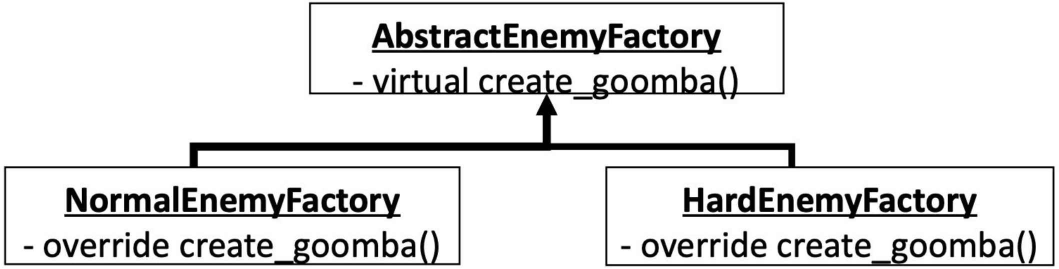
Creational patterns: abstract factories

- The ***abstract factory pattern*** encapsulates a group of factories that have a common theme without specifying their concrete classes.

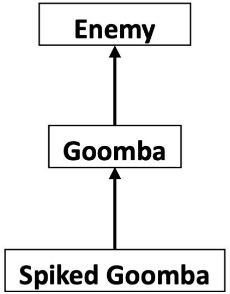
Creational patterns: abstract factories



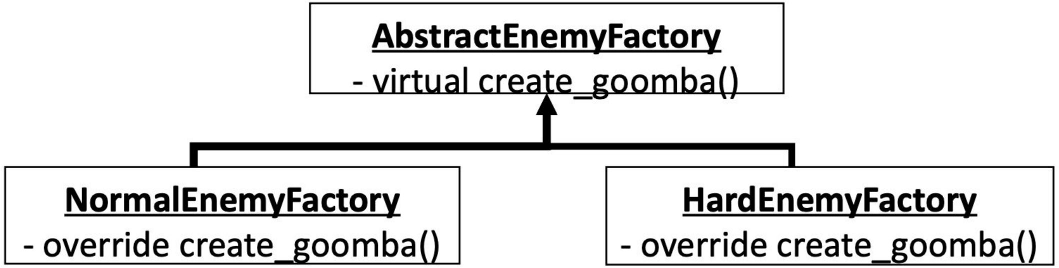
- The ***abstract factory pattern*** encapsulates a group of factories that have a common theme without specifying their concrete classes.



Creational patterns: abstract factories



- The ***abstract factory pattern*** encapsulates a group of factories that have a common theme without specifying their concrete classes.




```
// Only have to do this once!
AbstractEnemyFactory* factory = nullptr;
if (difficulty == “normal”)
  factory = new NormalEnemyFactory();
else if (difficulty == “hard”)
  factory = new HardEnemyFactory();
Enemy* goomba = factory->create_goomba();
```

Scenario: global application state

Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.

Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.
- The anti-pattern (**bad**) solution is to have an **unprotected global variable** (e.g., a public static field).

Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.
- The anti-pattern (**bad**) solution is to have an **unprotected global variable** (e.g., a public static field).
  - fails to control access or updates!

Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.
- The anti-pattern (**bad**) solution is to have an **unprotected global variable** (e.g., a public static field).
  - fails to control access or updates!
- A "less bad" solution is to put all of the state in one class and have a **global instance** of that class.

Scenario: global application state

- Global variables are usually a **poor design choice**. However:

Scenario: global application state

- Global variables are usually a **poor design choice**. However:
  - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. …)

Scenario: global application state

- Global variables are usually a **poor design choice**. However:
    - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. …)
        - This is not an argument for using global variables to avoid passing a few parameters.

Scenario: global application state

- Global variables are usually a **poor design choice**. However:
  - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. …)
    - This is not an argument for using global variables to avoid passing a few parameters.
  - Or if you need to access state stored outside your program (e.g., database, web API)

Scenario: global application state

- Global variables are usually a **poor design choice**. However:
  - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. …)
    - This is not an argument for using global variables to avoid passing a few parameters.
  - Or if you need to access state stored outside your program (e.g., database, web API)
  - Then global variables **may** be acceptable

Singleton design pattern

- The ***singleton pattern*** restricts the instantiation of a class to **exactly one** logical instance. It ensures that a class has only one logical instance at runtime and provides a global point of access to it.

Singleton

public:

- static ***get_instance***() *// named ctor*

private:

- static ***instance*** *// the one instance*
- Singleton() *// ctor*

Singleton design pattern: example

```
class Singleton {
 // public way to get “the one logical instance”
 public static Singleton get_instance() {
   if (Singleton.instance == null) Singleton.instance = new Singleton();
   return Singleton.instance;
 }
 private static Singleton instance = null;
 private Singleton() { // only runs once
   billing_database = 0;
   System.out.println("Singleton DB created");
 }
 // Our global state
 private int billing_database;
 public int get_billing_count() { return billing_database; }
 public void increment_billing_count() { billing_database += 1; }
}
```

Singleton design pattern: example

lazy initializaton
of single object

```
class Singleton {
 // public way to get “the one logical instance”
 public static Singleton get_instance() {
   if (Singleton.instance == null) Singleton.instance = new Singleton();
   return Singleton.instance;
 }
 private static Singleton instance = null;
 private Singleton() { // only runs once
   billing_database = 0;
   System.out.println("Singleton DB created");
 }
 // Our global state
 private int billing_database;
 public int get_billing_count() { return billing_database; }
 public void increment_billing_count() { billing_database += 1; }
}
```

Singleton design pattern: example

```
class Singleton {
 // public way to get "the one logical instance"
 public static Singleton get_instance() {
   if (Singleton.instance == null) Singleton.instance = new Singleton();
   return Singleton.instance;
 }
 private static Singleton instance = null;
 private Singleton() { // only runs once
   billing_database = 0;
   System.out.println("Singleton DB created");
 }
 // Our global state
 private int billing_database;
 public int get_billing_count() { return billing_database; }
 public void increment_billing_count() { billing_database += 1; }
}
```

this constructor can't be called any other way

Singleton design pattern: example

```
class Singleton {
 // public way to get "the one logical instance"
 public static Singleton get_instance() {
   if (Singleton.instance == null) Singleton.instance = new Singleton();
   return Singleton.instance;
 }
 private static Singleton instance = null;
 private Singleton() { // only runs once
   billing_database = 0;
   System.out.println("Singleton DB created");
 }
 // Our global state
 private int billing_database;
 public int get_billing_count() { return billing_database; }
 public void increment_billing_count() { billing_database += 1; }
}
```

all clients share this global state

Singleton design pattern: example

What is the output of this code?

<u>Singleton</u>
public:
- static ***get_instance***() *// named ctor*
- get_billing_count()
- increment_billing_count() *// adds 1*

private:
- static ***instance*** *// the one instance*
- Singleton() *// ctor, prints message*
- billing_database

```
class Main {
 public static void main(String[] args) {
   int bills = Singleton.get_instance().get_billing_count();
   System.out.println(bills);

   Singleton.get_instance().increment_billing_count();
   bills = Singleton.get_instance().get_billing_count();
   System.out.println(bills);
 }
}
```

Singleton design pattern: example

What is the output of this code?

<u>Singleton</u>

public:

- static ***get_instance***() *// named ctor*
- get_billing_count()
- increment_billing_count() *// adds 1*

private:

- static ***instance*** *// the one instance*
- Singleton() *// ctor, prints message*
- billing_database

```
class Main {
 public static void main(String[] args) {
   int bills = Singleton.get_instance().get_billing_count();
   System.out.println(bills);

   Singleton.get_instance().increment_billing_count();
   bills = Singleton.get_instance().get_billing_count();
   System.out.println(bills);
 }
}
```

Output:

```
Singleton DB created
0
1
```

Singleton design pattern: another example

- Suppose we are implementing a computer version of the card game Euchre. In addition to a few abstract datatypes, we have a Game class that stores the state needed for a game of Euchre. When started, our application prototype plays one game of Euchre and then exits.

- Design question: **should we make Game a singleton**?

Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application

Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application
- However, there only **happens** to be one instance of Game. There's **no requirement** that we only have one instance.

Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application
- However, there only **happens** to be one instance of Game. There's **no requirement** that we only have one instance.
- We should only use the Singleton pattern when current or future **requirements** dictate that only one instance should exist.

Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application
- However, there only **happens** to be one instance of Game. There's **no requirement** that we only have one instance.
- We should only use the Singleton pattern when current or future **requirements** dictate that only one instance should exist.
  - Singleton is **not** a license to make everything global.

Behavioural Design Patterns

Behavioural Design Patterns

- ***Behavioral design patterns*** support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.

Behavioural Design Patterns

- ***Behavioral design patterns*** support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.
  - Commonly used to enable **limited sharing**

Behavioural Design Patterns

- ***Behavioral design patterns*** support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.
  - Commonly used to enable **limited sharing**
    - e.g., same underlying algorithm, different interfaces or same interface, different underlying algorithms

Behavioural Design Patterns

- ***Behavioral design patterns*** support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.
  - Commonly used to enable **limited sharing**
    - e.g., same underlying algorithm, different interfaces or same interface, different underlying algorithms
  - Examples: strategy pattern, template method pattern, iterator pattern, observer pattern, etc.

Iterator Pattern

- The ***iterator pattern*** is a common behavioral design pattern. It provides a uniform interface for traversing containers regardless of how they are implemented.

Iterator Pattern

- The ***iterator pattern*** is a common behavioral design pattern. It provides a uniform interface for traversing containers regardless of how they are implemented.
    - e.g., Java's List interface doesn't care whether it's backed by an array or a linked list

Iterator Pattern

- The ***iterator pattern*** is a common behavioral design pattern. It provides a uniform interface for traversing containers regardless of how they are implemented.
  - e.g., Java's List interface doesn't care whether it's backed by an array or a linked list
- Similar patterns exist for other kinds of data structures
  - e.g., ***visitor pattern*** for tree-like structures

Strategy Design Pattern

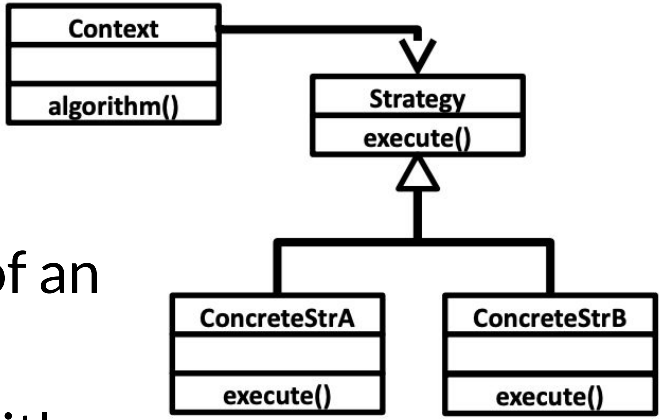
Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm

Strategy Design Pattern

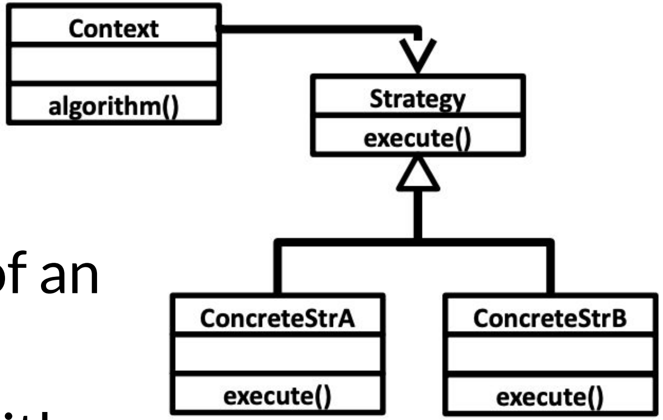
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm

Strategy Design Pattern



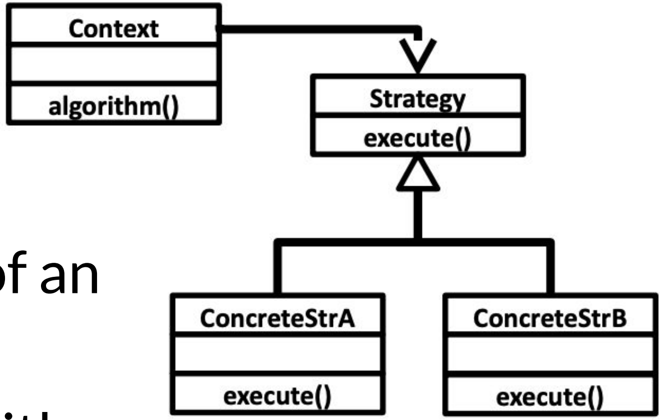
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm

Strategy Design Pattern



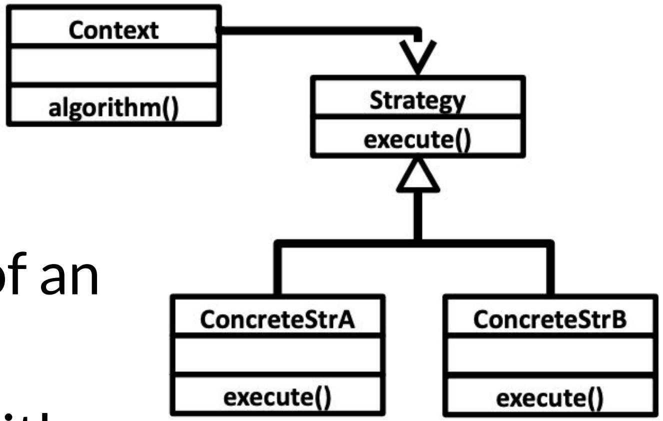
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:

Strategy Design Pattern



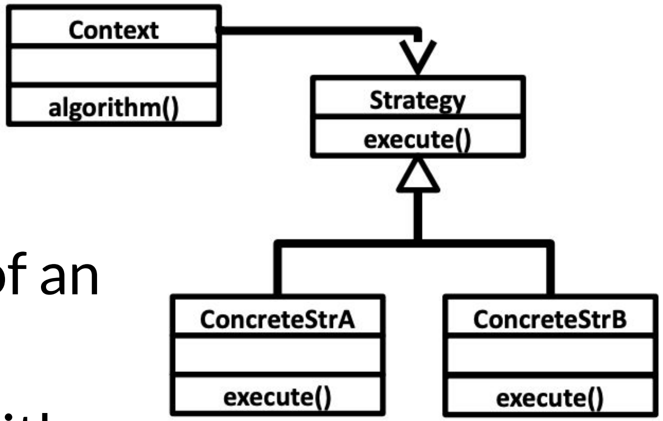
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations

Strategy Design Pattern



- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context

Strategy Design Pattern



- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces extra interfaces and classes: code can be harder to understand; adds overhead if the strategies are simple

Template Method Design Pattern

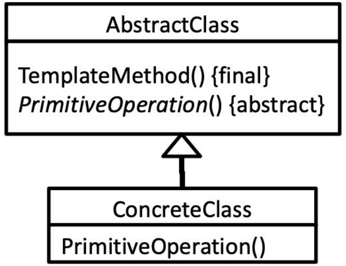
Template Method Design Pattern

- Problem: An algorithm has **customizable** and **invariant** parts

Template Method Design Pattern

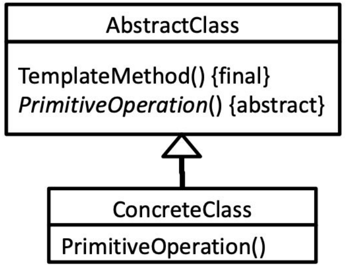
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.

Template Method Design Pattern



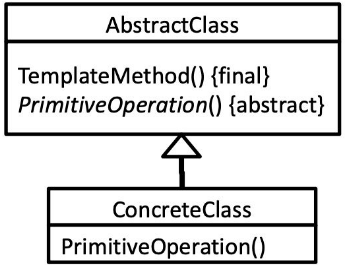
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.

Template Method Design Pattern



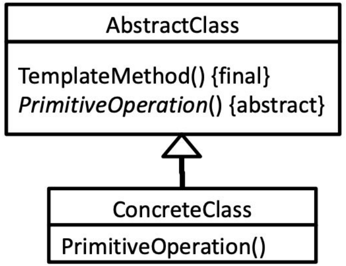
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:

Template Method Design Pattern



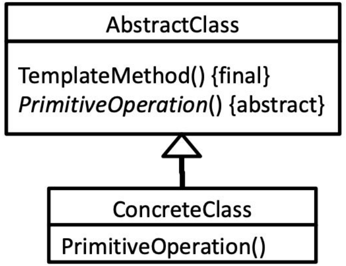
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
  - Code reuse for the invariant parts of algorithm

Template Method Design Pattern



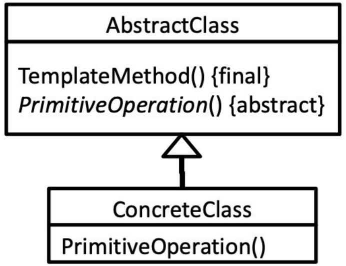
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
    - Code reuse for the invariant parts of algorithm
    - Customization is restricted to the primitive operations

Template Method Design Pattern



- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
  - Inverted (“Hollywood-style”) control for customization: “don’t call us, we’ll call you” (cf. comparison function in sorting)

Template Method Design Pattern



- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
    - Code reuse for the invariant parts of algorithm
    - Customization is restricted to the primitive operations
    - Inverted ("Hollywood-style") control for customization: "don't call us, we'll call you" (cf. comparison function in sorting)
    - Invariant parts of the algorithm are not changed by subclasses

Template vs. Strategy Design Pattern

Template vs. Strategy Design Pattern

- Both support variation in a larger context

Template vs. Strategy Design Pattern

- Both support variation in a larger context
- **Template method** uses inheritance + an overridable method

Template vs. Strategy Design Pattern

- Both support variation in a larger context
- **Template method** uses inheritance + an overridable method
- **Strategy** uses an interface and polymorphism (via composition)
    - Strategy objects are reusable across multiple classes
    - Multiple strategy objects are possible per class

Scenario: binge-watching

- Suppose we're implementing a video streaming website in which users can “binge-watch” (or “lock on”) to one channel. The user will then see that channel's videos in sequence. When the last such video is watched, the user should stop binge-watching that channel.

Scenario: binge-watching

- Idea: when the last video is watched, call release_binge_watch() on the user.

Scenario: binge-watching

- Idea: when the last video is watched, call release_binge_watch() on the user.

```
class User {
 public void release_binge_watch(Channel c) {
   if (c == binge_channel) {
     binge_channel = null;
   }
 }
 private Channel binge_channel;
}
```

Scenario: binge-watching

- Idea: when the last video is watched, call release_binge_watch() on the user.

```
class User {
 public void release_binge_watch(Channel c) {
   if (c == binge_channel) {
     binge_channel = null;
   }
 }
 private Channel binge_channel;
}
```

```
class Channel {
 // Called when the last video is shown
 public void on_last_video_shown() {
   // Global accessor for the user
   get_user().release_binge_watch(this);
 }
}
```

Scenario: binge-watching

- Idea: when the last video is watched, call release_binge_watch() on the user.

```
class User {
 public void release_binge_watch(Channel c) {
   if (c == binge_channel) {
     binge_channel = null;
   }
 }
 private Channel binge_channel;
}
```

```
class Channel {
 // Called when the last video is shown
 public void on_last_video_shown() {
   // Global accessor for the user
   get_user().release_binge_watch(this);
 }
}
```

- What are some problems with this approach?

Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
    - Changing one likely requires a change to the other

Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
    - Changing one likely requires a change to the other
- The design does not support multiple users

Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other
- The design does not support multiple users
- What if we later want to update a user's "recommendation queue" when they finish binge-watching a channel?

Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other
- The design does not support multiple users
- What if we later want to update a user's “recommendation queue” when they finish binge-watching a channel?
- Whenever requirements change and we want to do something else when a video finishes (e.g., update advertising) we **must update the Channel class** and couple it to the new feature

Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other
- The design does r
- What if we later w mendation queue” when they finish binge-watching a channel?
- Whenever requirements change and we want to do something else when a video finishes (e.g., update advertising) we **must update the Channel class** and couple it to the new feature

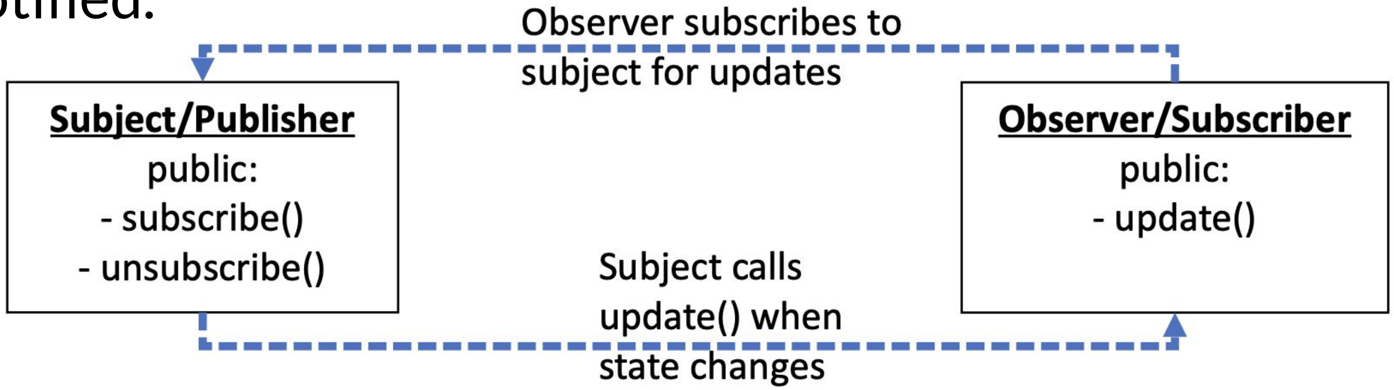
What can we do instead?

Observer Pattern

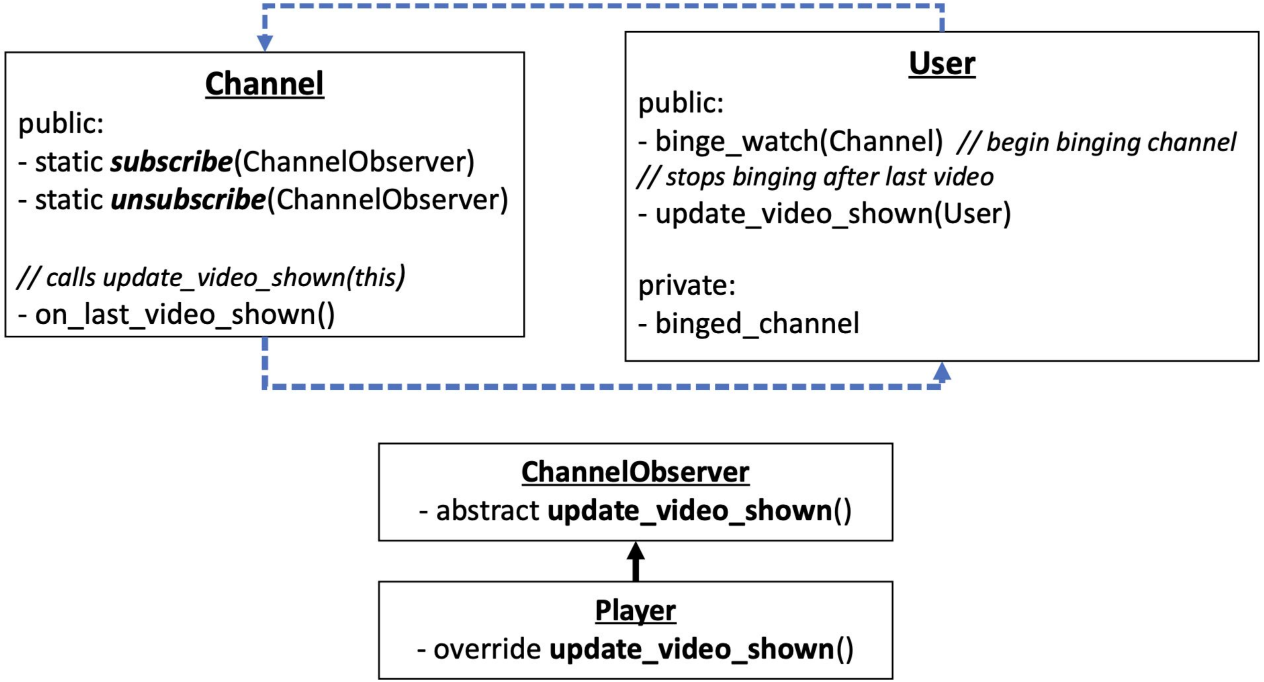
- The ***observer pattern*** (also called "***publish-subscribe***") allows dependent objects to be notified automatically when the state of a subject changes. It defines a one-to-many dependency between objects so that when one object changes state, all of it dependents are notified.

Observer Pattern

- The ***observer pattern*** (also called "***publish-subscribe***") allows dependent objects to be notified automatically when the state of a subject changes. It defines a one-to-many dependency between objects so that when one object changes state, all of it dependents are notified.

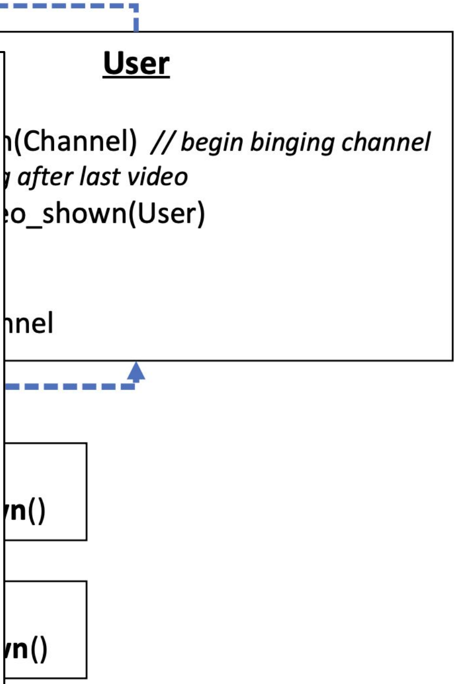


Observer Pattern: bing-watch scenario



Observer Pattern: bing-watch scenario

```
class Channel {
 public static void subscribe(ChannelObserver obs) {
   subscribers.Add(obs);
 }
 public static void unsubscribe(ChannelObserver obs) {
   subscribers.Remove(obs);
 }
 public void on_last_video_shown() {
   foreach (ChannelObserver obs in subscribers) {
     observer.update_video_shown(this);
   }
 }
 private static List<ChannelObserver> subscribers =
          new List<ChannelObserver>();
}
```



Observer Pattern: bing-watch scenario

```
interface ChannelObserver {
 void update_video_shown(Channel channel);
}
```

```
class Channel {
 public static void subscribe(ChannelObserver obs) {
   subscribers.Add(obs);
 }
 public static void unsubscribe(ChannelObserver obs) {
   subscribers.Remove(obs);
 }
 public void on_last_video_shown() {
   foreach (ChannelObserver obs in subscribers) {
     observer.update_video_shown(this);
   }
 }
 private static List<ChannelObserver> subscribers =
          new List<ChannelObserver>();
}
```

(Channel) *// begin binging channel*
after last video
o_shown(User)
nnel
n()
n()

Observer Pattern: bing-watch scenario

```
interface ChannelObserver {
 void update_video_shown(Channel channel);
}
```

(Channel) *// begin binging channel*

after last video

o_shown(User)

nnel

```
class Channel {
 public static void subscribe(ChannelObserver obs) {
   subscribers.Add(obs);
 }
 public static void unsubscribe(ChannelObserver obs) {
   subscribers.Remove(obs);
 }
 public void on_last_video_shown() {
   foreach (ChannelObserver obs in subscribers) {
     observer.update_video_shown(this);
   }
 }
 private static List<ChannelObserver> subscribers =
          new List<ChannelObserver>();
}
```

```
class User: ChannelObserver {
 public void update_video_shown(Channel c) {
   if (c == binged_channel)
     binged_channel = null;
 }
 public void binge_watch(Channel c) {
   binged_channel = c;
 }
 private Channel binged_channel;
}
```

Observer Pattern: update functions

- Having multiple "update_" functions, one for each type of state change, keeps messages **granular**

Observer Pattern: update functions

- Having multiple “update_” functions, one for each type of state change, keeps messages **granular**
  - Observers that do not care about a particular type of update can ignore it (via an empty implementation of the update function)

Observer Pattern: update functions

- Having multiple "update_" functions, one for each type of state change, keeps messages **granular**
  - Observers that do not care about a particular type of update can ignore it (via an empty implementation of the update function)
- Generally it is better to pass the newly-updated data as a parameter to the update function (**push**) as opposed to making observers fetch it each time (**pull**)

Design patterns: takeaways

- Thinking about design before you start coding is usually worthwhile for large projects
  - Design around the most expensive parts of the software engineering process (usually maintainence!)
- Design patterns are re-usable solutions to common problems
- Be familiar with them enough to recognize when they're being used
  - and to know when to use them yourself
  - you can look up details of a pattern if you remember its name!
- Be mindful of and avoid common anti-patterns

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

Today's agenda:

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

Technical debt

Technical debt

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

Technical debt

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

- analogy to **financial debts**:

Technical debt

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

- analogy to **financial debts**:
  - you gain some immediate benefit
    - in a financial debt, you gain a large sum of money
    - in a technical debt, you gain implementation speed, etc.

Technical debt

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

- analogy to **financial debts**:
  - you gain some immediate benefit
    - in a financial debt, you gain a large sum of money
    - in a technical debt, you gain implementation speed, etc.
  - you pay for it over time
    - in a financial debt, you pay interest
    - in a technical debt, your maintenance costs increase

Technical debt: benefits

- Why might you **intentionally** make a sub-optimal design decision?

Technical debt: benefits

- Why might you **intentionally** make a sub-optimal design decision?
  - Cost
    - either in dev time or because the code isn't done yet
  - Need to meet a deadline
  - Avoid premature optimization
  - Code reuse
  - Principle of least surprise
  - Organizational requirements/politics
  - etc.

Technical debt: paying interest

- Unlike a financial debt, a technical debt doesn't have a **creditor**

Technical debt: paying interest

- Unlike a financial debt, a technical debt doesn’t have a **creditor**
  - Conceptually, when you take on technical debt you are borrowing from **future maintainers** of the system

Technical debt: paying interest

- Unlike a financial debt, a technical debt doesn't have a **creditor**
  - Conceptually, when you take on technical debt you are borrowing from **future maintainers** of the system
- Recall our goals in good design:

Technical debt: paying interest

- Unlike a financial debt, a technical debt doesn't have a **creditor**
  - Conceptually, when you take on technical debt you are borrowing from **future maintainers** of the system
- Recall our goals in good design:
  - design for **change and reuse**
  - make the system easy to extend, modify, etc.

Technical debt: paying interest

- Unlike a financial debt, a technical debt doesn't have a **creditor**
  - Conceptually, when you take on technical debt you are borrowing from **future maintainers** of the system
- Recall our goals in good design:
  - design for **change and reuse**
  - make the system easy to extend, modify, etc.
- **Implication**: a system with technical debt is **harder** to change and reuse

Technical debt: benefits and costs

Examples of debt:

Examples of costs:

Technical debt: benefits and costs

Examples of debt:

- code smells

Examples of costs:

Technical debt: benefits and costs

Examples of debt:

- code smells

Examples of costs:

- “smelly” code is less flexible

Technical debt: benefits and costs

Examples of debt:

- code smells
- missing tests

Examples of costs:

- “smelly” code is less flexible

Technical debt: benefits and costs

Examples of debt:

- code smells
- missing tests

Examples of costs:

- “smelly” code is less flexible
- tests don’t catch breaking change, causing outages

Technical debt: benefits and costs

Examples of debt:

- code smells
- missing tests
- missing documentation

Examples of costs:

- “smelly” code is less flexible
- tests don’t catch breaking change, causing outages

Technical debt: benefits and costs

Examples of debt:

- code smells
- missing tests
- missing documentation

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

Technical debt: benefits and costs

Examples of debt:

- code smells
- missing tests
- missing documentation
- dependency on old versions of third-party systems

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

Technical debt: benefits and costs

Examples of debt:

- code smells
- missing tests
- missing documentation
- dependency on old versions of third-party systems

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

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

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, refactoring, and maintenance (2/2)

Martin Kellogg

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
  - You **must s**
  - You **do not**
    you're read

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 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** 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

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!)

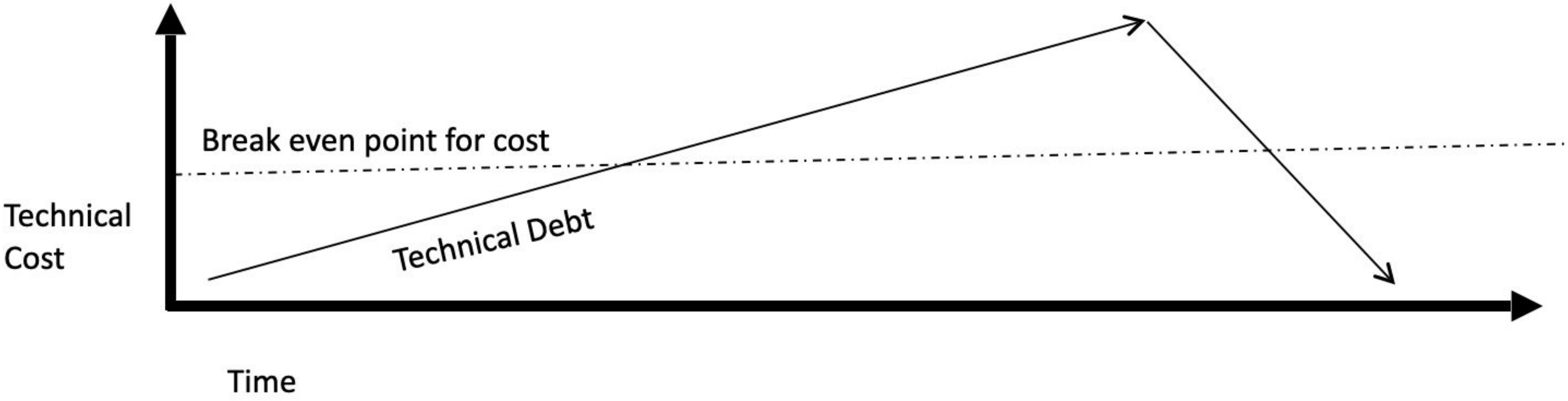
Paying down technical debt

- 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

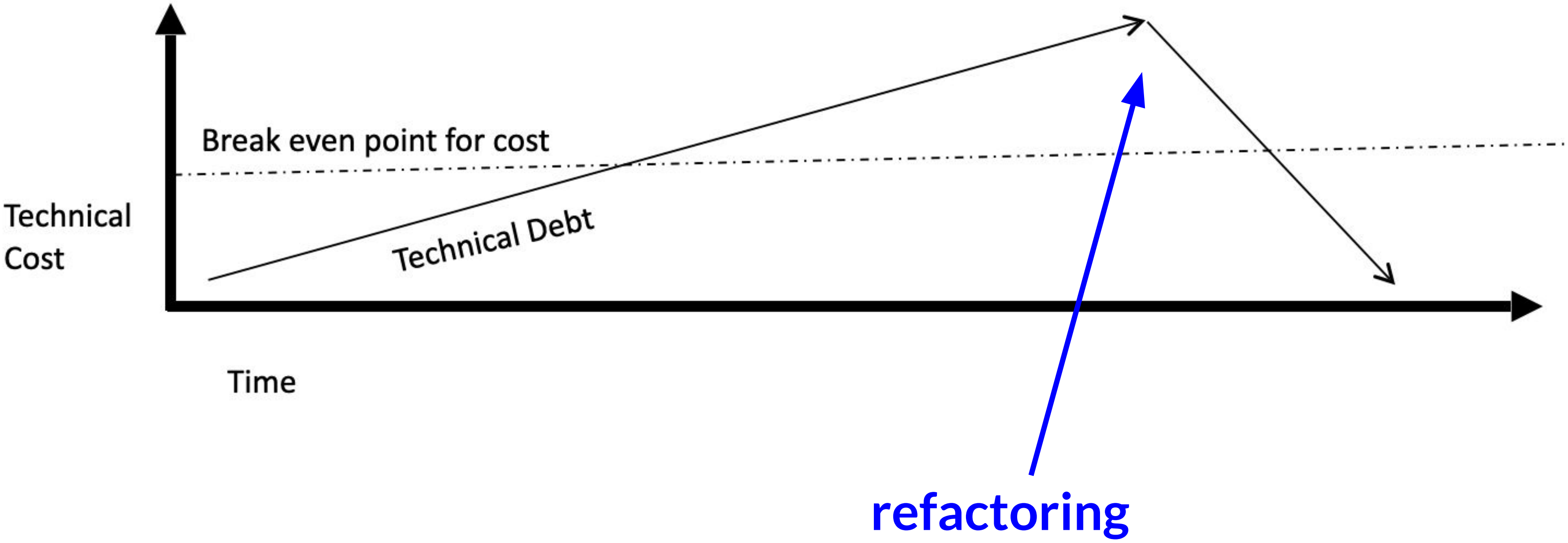
Paying down technical debt

- 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
- 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”

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 (1): reading quiz

Tech debt (1): reading quiz

Q1: Which of the following was used as an example of a system where technical debt was taken on intentionally, but a serious consequence occurred that made the debt not worthwhile?

A. Firefox at Mozilla
B. the Mars Polar Orbiter at NASA
C. CalMail at Berkeley

Q2: **TRUE** or **FALSE**: the author argues that there are “some forms of technical debt are so expensive that they should be avoided entirely whenever possible”

Tech debt (1): reading quiz

Q1: Which of the following was used as an example of a system where technical debt was taken on intentionally, but a serious consequence occurred that made the debt not worthwhile?

A. Firefox at Mozilla
B. the Mars Polar Orbiter at NASA
C. CalMail at Berkeley

Q2: **TRUE** or **FALSE**: the author argues that there are “some forms of technical debt are so expensive that they should be avoided entirely whenever possible”

Tech debt (1): reading quiz

Q1: Which of the following was used as an example of a system where technical debt was taken on intentionally, but a serious consequence occurred that made the debt not worthwhile?

A. Firefox at Mozilla
B. the Mars Polar Orbiter at NASA
C. CalMail at Berkeley

Q2: TRUE or **FALSE**: the author argues that there are "some forms of technical debt are so expensive that they should be avoided entirely whenever possible"

Tech debt (2): reading quiz

Tech debt (2): reading quiz

Q1: Which of these does one of the authors identify as a "fundamental law of programming"?

A. it's harder to read code than to write it
B. testing can show the presence of bugs, but not their absence
C. always double your estimates (even those you already doubled)

Q2: The main complaint of the author of "software disenchantment" is:

A. programmers are lazier today than they used to be
B. modern software uses more resources to do the same job
C. his manager makes unreasonable demands that lead to tech debt

Tech debt (2): reading quiz

Q1: Which of these does one of the authors identify as a "fundamental law of programming"?

A. it's harder to read code than to write it
B. testing can show the presence of bugs, but not their absence
C. always double your estimates (even those you already doubled)

Q2: The main complaint of the author of "software disenchantment" is:

A. programmers are lazier today than they used to be
B. modern software uses more resources to do the same job
C. his manager makes unreasonable demands that lead to tech debt

Tech debt (2): reading quiz

Q1: Which of these does one of the authors identify as a "fundamental law of programming"?

A. it's harder to read code than to write it
B. testing can show the presence of bugs, but not their absence
C. always double your estimates (even those you already doubled)

Q2: The main complaint of the author of "software disenchantment" is:

A. programmers are lazier today than they used to be
B. modern software uses more resources to do the same job
C. his manager makes unreasonable demands that lead to tech debt

Critical discussion of “software disenchantment”

Critical discussion of “software disenchantment”

We had a brief discussion of the second reading for today. Highlights:

- the author is mostly complaining, and doesn’t actually offer solutions. This is very common (but not very productive!)
- this sort of article is popular with a certain segment of engineers (e.g., HackerNews). You will hear these kind of hot takes again
  - be ready to recognize this when you run into it
- I (Prof Martin) don’t agree with all of these complaints
- One explanation for “why is software so big” is that it handles many special cases well (i.e., many bugs have already been fixed)
  - rewriting would destroy this collective knowledge!