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

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:

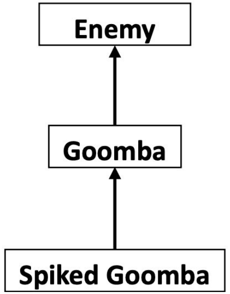
```
Enemy* goomba = nullptr;
if (difficulty == "normal")
  goomba = new Goomba();
else if (difficulty == "hard")
  goomba = new SpikedGoomba();
```

Why is this bad?
- code duplication
- consider how you'd add a new difficulty level...

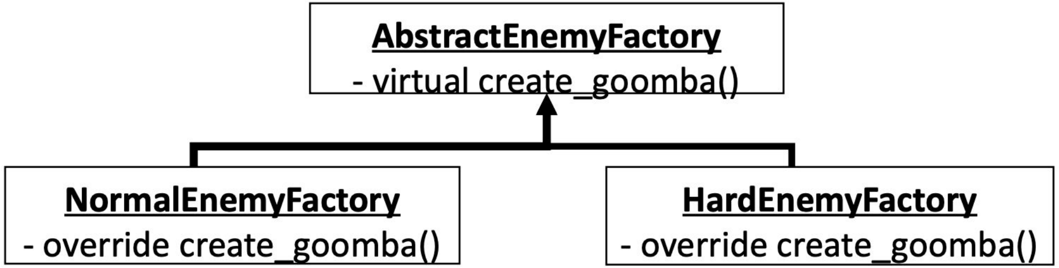
Creational patterns: abstract factories

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

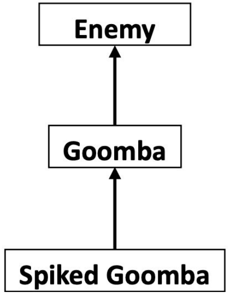
Creational patterns: abstract factories



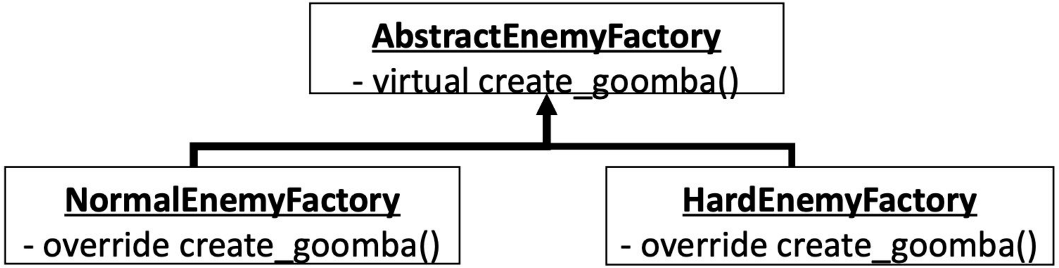
- The ***abstract factory pattern*** encapsulates a group of factories that have a common theme without specifying their concrete classes.



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

```
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; }
}
```

lazy initializaton of single object

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?

Singleton

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?

Singleton

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: get_instance()

- Could we avoid typing Single.get_instance() so many times by doing this at all of the points in our program that use the singleton?

```
Single s = Singleton.get_instance();
System.out.println(s.get_billing_count());
… // later
System.out.println(s.get_billing_count());
```

Singleton design pattern: get_instance()

- Could we avoid typing Single.get_instance() so many times by doing this at all of the points in our program that use the singleton?

```
Single s = Singleton.get_instance();
System.out.println(s.get_billing_count());
… // later
System.out.println(s.get_billing_count());
```

- Is this a good idea or not?

Singleton design pattern: get_instance()

- Could we avoid typing Single.get_instance() so many times by doing this at all of the points in our program that use the singleton?

```
Single s = Singleton.get_inst
System.out.println(s.get_bill
… // later
System.out.println(s.get_bill
```

- Is this a good idea or not?

This is a **bad idea**. There is **no guarantee** that get_instance() will return the same pointer (same object) every time it is called. (It may return different **concrete copies** of the **same logical item**.)

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

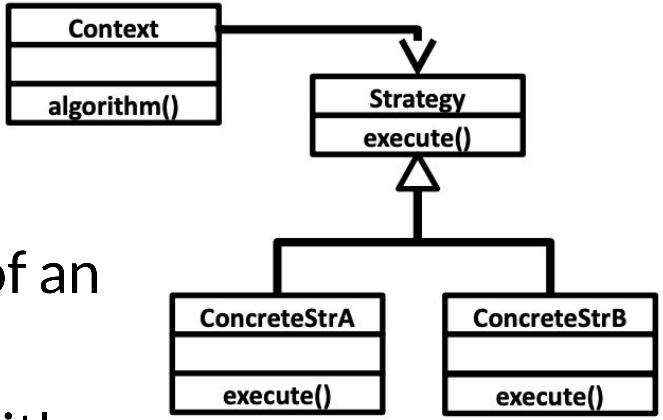
Strategy Design Pattern

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

Strategy Design Pattern

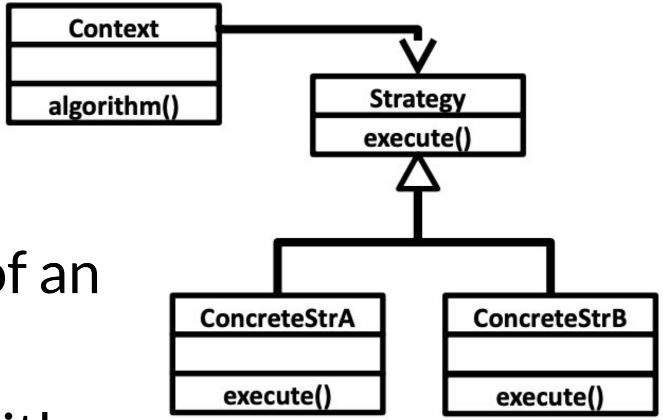
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm

Strategy Design Pattern



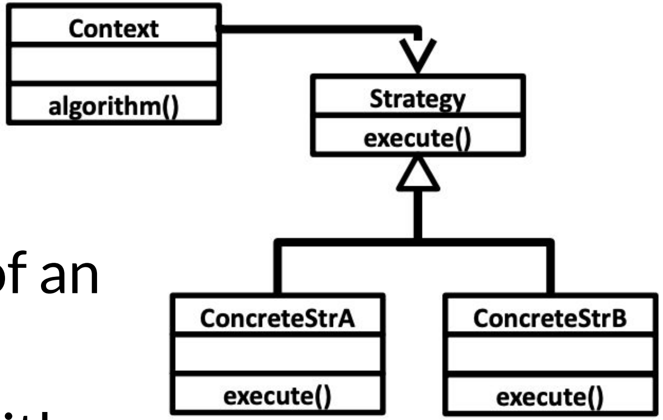
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm

Strategy Design Pattern



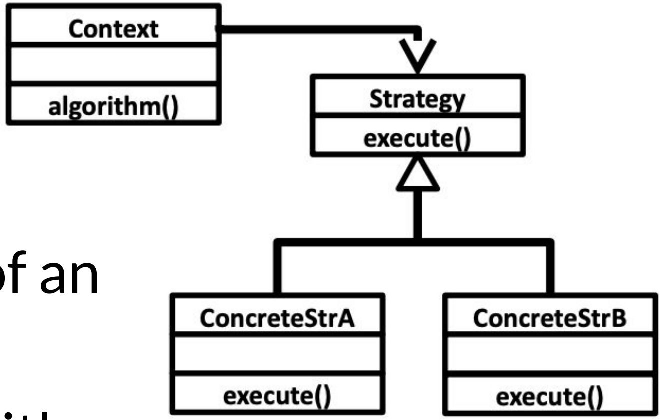
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:

Strategy Design Pattern



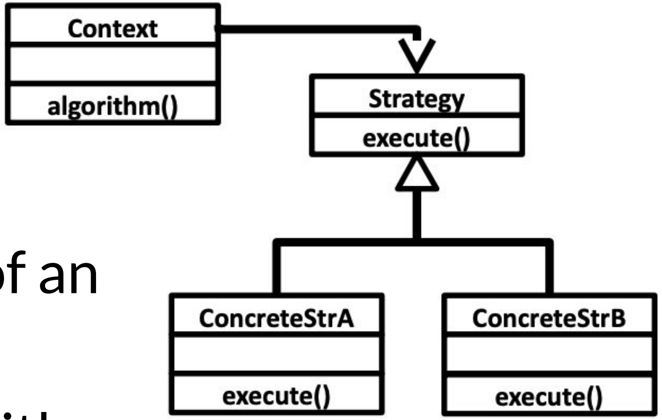
- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations

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

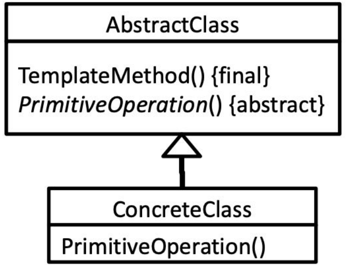
Template Method Design Pattern

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

Template Method Design Pattern

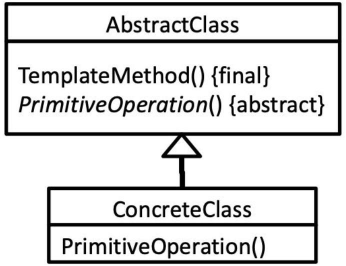
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.

Template Method Design Pattern



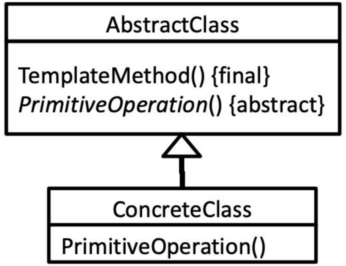
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.

Template Method Design Pattern



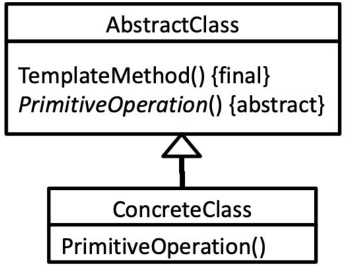
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:

Template Method Design Pattern



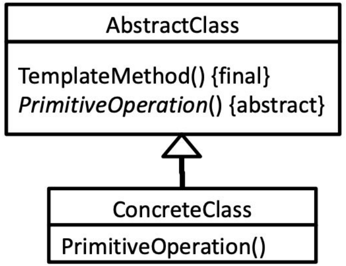
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences: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

Template Method Design Pattern



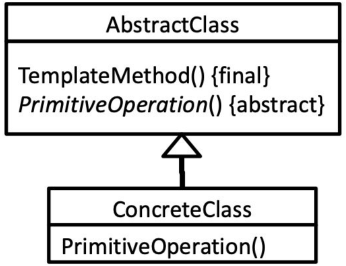
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences: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

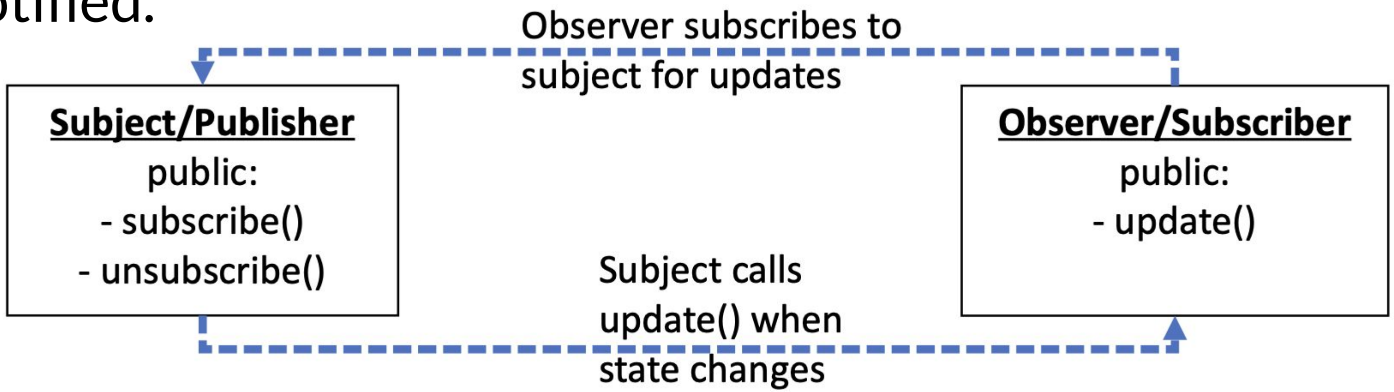
What can we do instead?

Observer Pattern

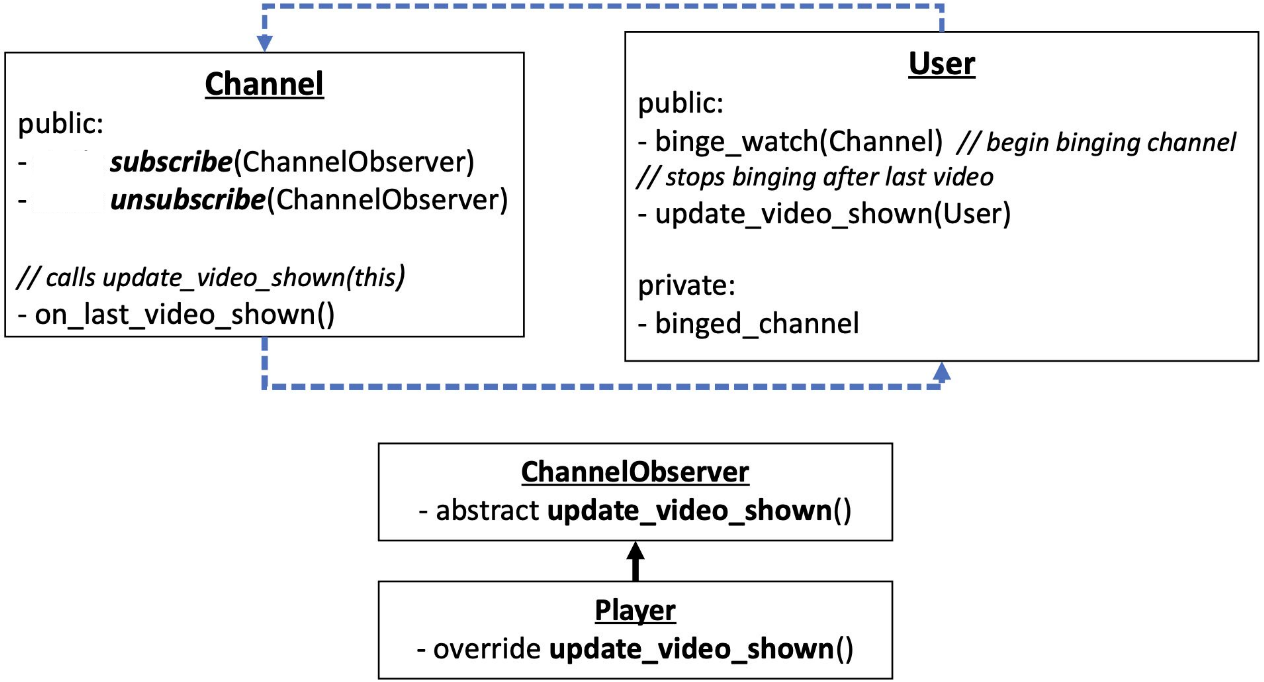
- The ***observer pattern*** (also called "***publish-subscribe***") allows dependent objects to be notified automatically when the state of a subject changes. It defines a one-to-many dependency between objects so that when one object changes state, all of it dependents are notified.

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.

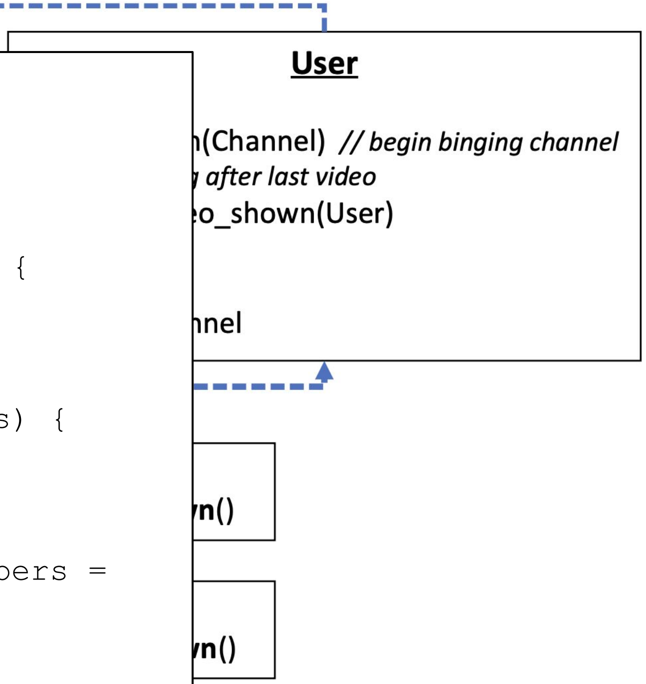


Observer Pattern: bing-watch scenario



Observer Pattern: bing-watch scenario

```
class Channel {
 public void subscribe(ChannelObserver obs) {
   subscribers.Add(obs);
 }
 public 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 void subscribe(ChannelObserver obs) {
   subscribers.Add(obs);
 }
 public 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>();
}
```

n(Channel) *// begin binging channel*
g after last video
eo_shown(User)

nnel

wn()

wn()

Observer Pattern: bing-watch scenario

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

```
class Channel {
 public void subscribe(ChannelObserver obs) {
   subscribers.Add(obs);
 }
 public 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>();
}
```

ı(Channel) *// begin binging channel*

ı after last video

ɛo_shown(User)

ınel

```
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

Reading quiz: technical debt

Reading quiz: technical debt

Q1: **TRUE** or **FALSE**: the article argued that it is both possible and desirable to avoid technical debt entirely.

Q2: The cost of taking on a financial debt is interest. The cost of taking on technical debt is increased ____________________ costs.

Reading quiz: technical debt

Q1: **TRUE** or **FALSE**: the article argued that it is both possible and desirable to avoid technical debt entirely.

Q2: The cost of taking on a financial debt is interest. The cost of taking on technical debt is increased ____________________ costs.

Reading quiz: technical debt

Q1: **TRUE** or **FALSE**: the article argued that it is both possible and desirable to avoid technical debt entirely.

Q2: The cost of taking on a financial debt is interest. The cost of taking on technical debt is increased ___maintanence____ costs.

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: 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!**