Introduction to Testing and Maintainable code

Reasons not to write unit tests

1. “I don't know how to write tests.”
2. “Writing tests is too hard.”
3. “I don't have enough time to write tests.”
4. “Testing is not my job.”
5. “My code is too simple for tests.”
6. “My code is too compex for tests.”

Reasons to write unit tests

1. Tests reduce bugs in new and existing features
2. Tests are good documentation of requirements
3. Tests reduce the cost of change
4. Tests improve design
5. Tests allow fearless refactoring

What should be tested?

- Unit tests are cheap and fast and there should be many
- Integration tests might require extensive setup of external systems and thus are not as fast as pure unit tests
- GUI tests are complex to be maintained automatically

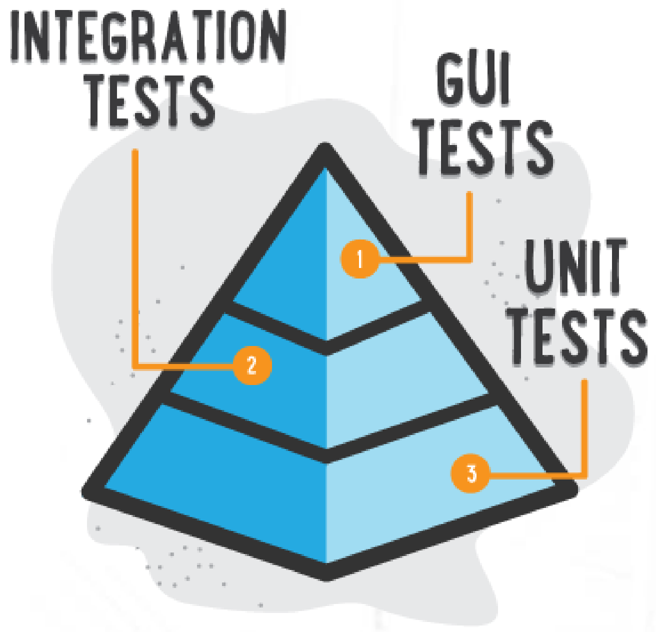
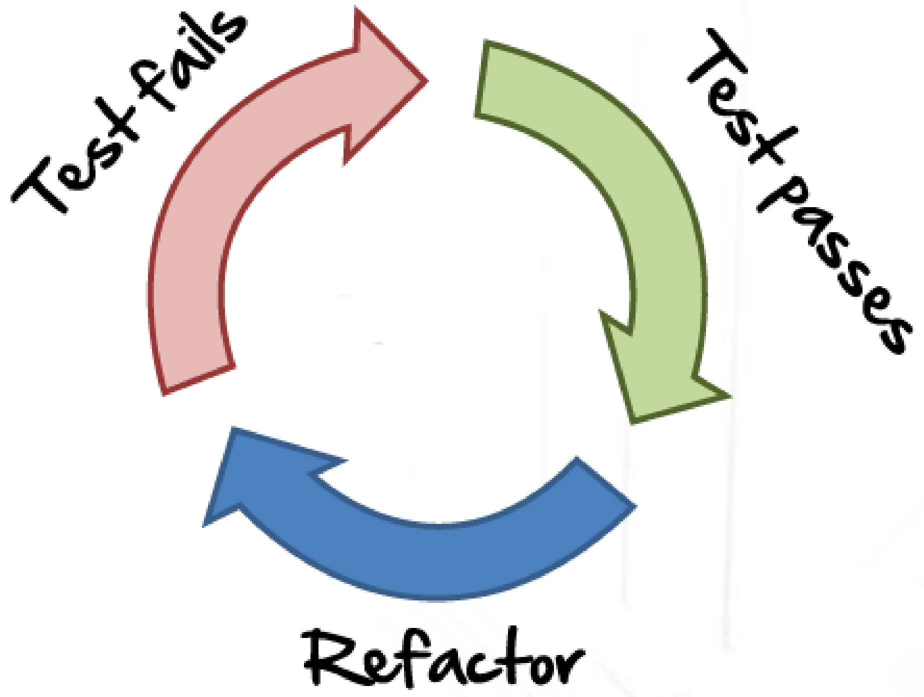


ILLUSTRATION BY SEGUE TECHNOLOGIES

What is TDD?

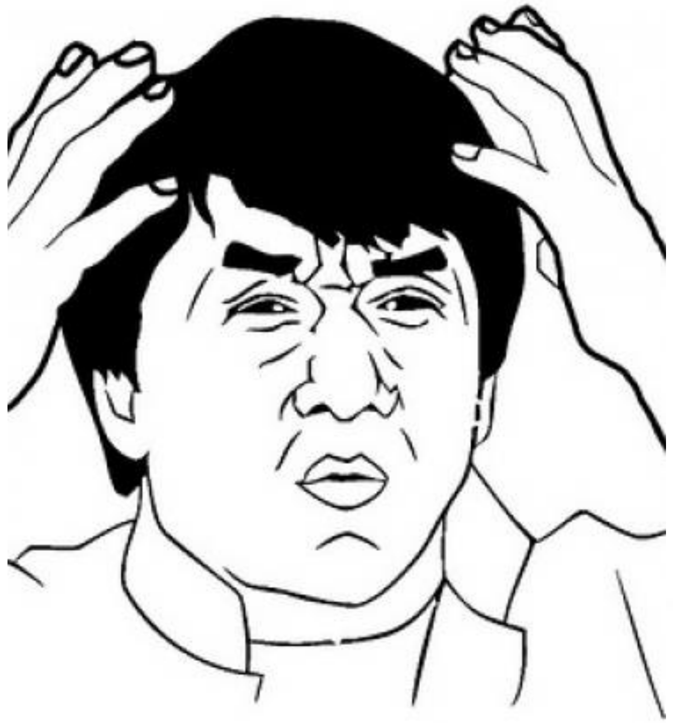
- Test Driven Development

- A continuous process
  - Write tests early
  - Run tests often
  - Get prompt feedback



But where do I start?

- Using the right tools
  - Unit testing framework (DUnit, DUnit2, DUnitX)
  - IDE Integration (TestInsight)
  - Continuous integration (FinalBuilder, Continua, ...)
  - FastMM, CodeCoverage, LeakCheck, ...
- Making code testable
  - Follow design patterns and coding principles
  - Clean code, SOLID, ...

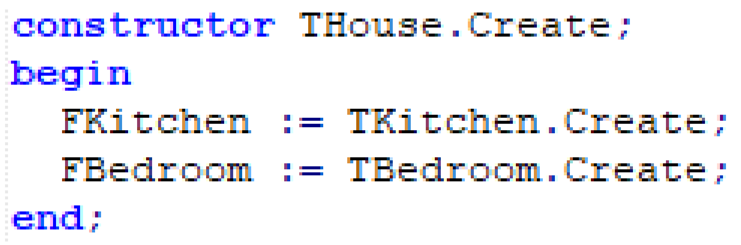


What is testability?

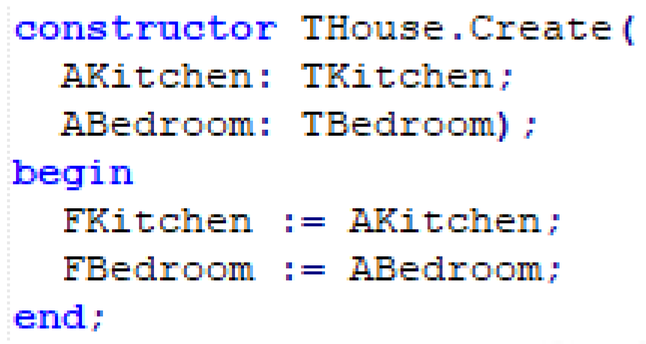
What is testable code?

- „Seams“ to separate different code pieces from each other
- No hidden input in methods
- Avoid singletons, other global state and hardwired external systems (database, filesystem, ...)
- Use principles like SOLID

Constructor does real work



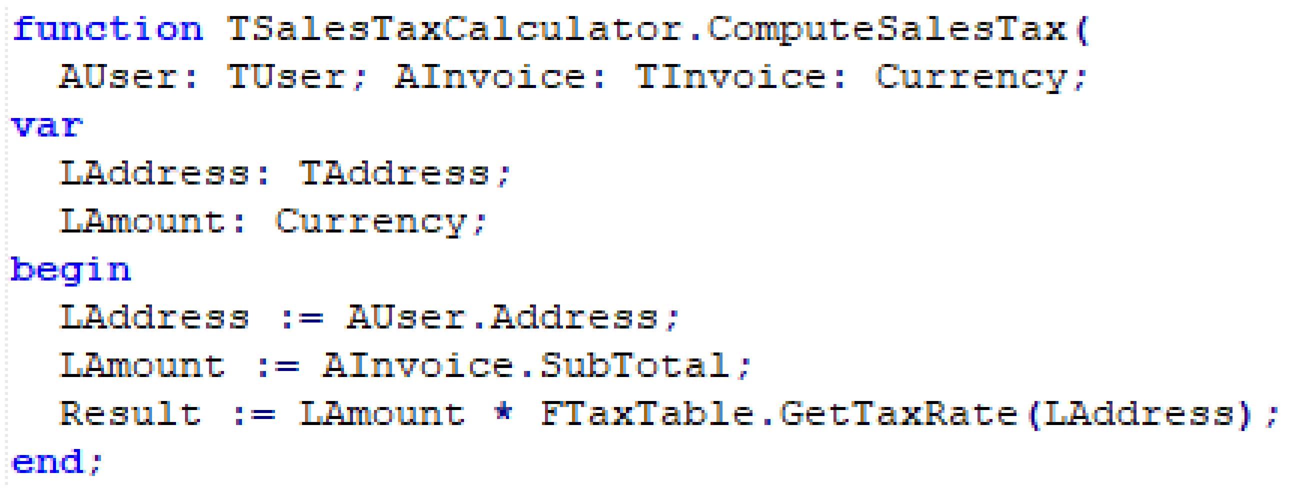
```
constructor THouse.Create;
begin
  FKitchen := TKitchen.Create;
  FBedroom := TBedroom.Create;
end;
```



```
constructor THouse.Create(
  AKitchen: TKitchen;
  ABedroom: TBedroom);
begin
  FKitchen := AKitchen;
  FBedroom := ABedroom;
end;
```

How to fix: use dependency injection

Service object digging in value object



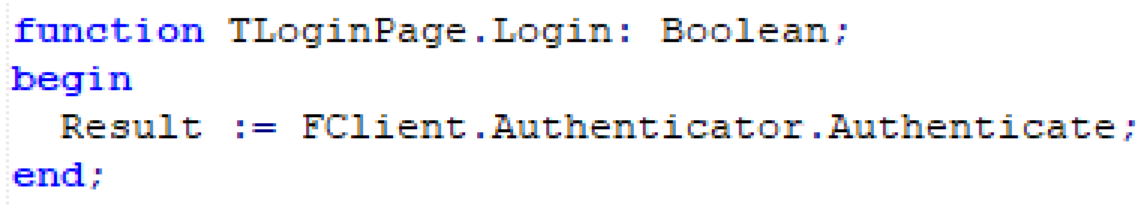
```
function TSalesTaxCalculator.ComputeSalesTax(
  AUser: TUser; AInvoice: TInvoice: Currency;
var
  LAddress: TAddress;
  LAmount: Currency;
begin
  LAddress := AUser.Address;
  LAmount := AInvoice.SubTotal;
  Result := LAmount * FTaxTable.GetTaxRate(LAddress);
end;
```

How to fix: directly pass what is needed

Service object digging in value object

```
function TSalesTaxCalculator.ComputeSalesTax(
  AAddress: TAddress; AAmount: Currency): Currency;
begin
  Result := AAmount * FTaxTable.GetTaxRate(AAddress);
end;
```

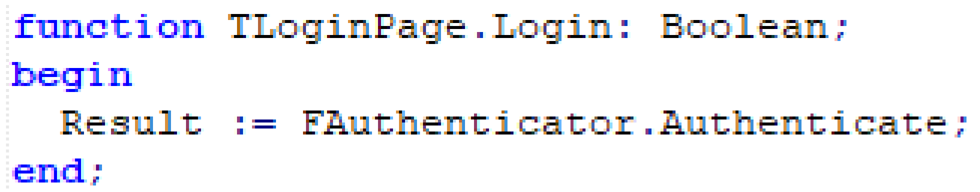
Service directly violating law of demeter



```
function TLoginPage.Login: Boolean;
begin
  Result := FClient.Authenticator.Authenticate;
end;
```

How to fix: again pass only what is directly needed

Service directly violating law of demeter



```
function TLoginPage.Login: Boolean;
begin
  Result := FAuthenticator.Authenticate;
end;
```

Global state and singleton

- Dirties the design
  - If A and B are not passed to each other they should not be able to interact
- Enables „spooky action at a distance“
  - Interaction with unknown collaborators
- Makes for brittle applications (and tests)
  - Changes in implementation may break them
- Turns APIs into liars

- How to fix?

Class does too much

- Why:
  - Hard to debug
  - Hard to test
  - Not or very hard to extend
  - Difficult to understand

- How to detect:
  - Try to sum up what a class does in a single phrase
  - ‚and‘, ‘or‘ are signs that it may do too much

Roads to clean code

1. Trial and error
   - Takes long time
   - May take you in completely wrong direction
2. Follow recommended practices
   - Many human-years went into them
   - Years of practical use removed the bad ideas
3. Write comments

Roads to clean code

1. Trial and error
   - Takes long time
   - May take you in completely wrong direction
2. Follow recommended practices
   - Many human-years went into them
   - Years of practical use removed the bad ideas
3. ~~Write comments~~

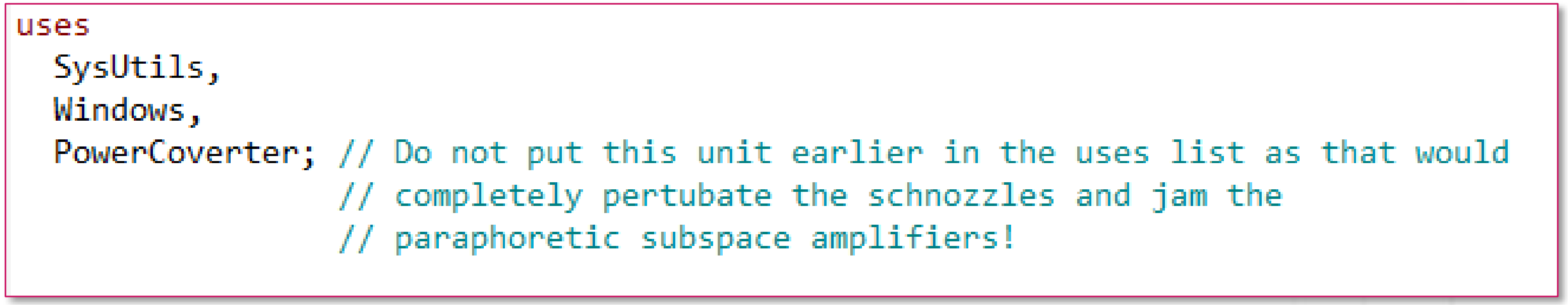
Commenting

- Writing good comments is incredibly hard
  - Comments are not for you, they are for unknown reader!
- If a code needs to be commented, it is too complicated
  - Exception: Document road <u>not</u> to be taken in the future!

Commenting

```
uses
  SysUtils,
  Windows,
  PowerCoverter; // Do not put this unit earlier in the uses list as that would
                 // completely pertubate the schnozzles and jam the
                 // paraphoretic subspace amplifiers!
```

Commenting



```
uses
  SysUtils,
  Windows,
  PowerCoverter; // Do not put this unit earlier in the uses list as that would
                 // completely pertubate the schnozzles and jam the
                 // paraphoretic subspace amplifiers!
```

Commenting

- Writing good comments is incredibly hard
  - Comments are not for you, they are for unknown reader!
- If a code needs to be commented, it is too complicated
  - Exception: Document road not to be taken in the future!
- Document APIs, not the code

Different levels of “pre-packaged” knowledge

- Design principles
  - High-level recommendations, not problem-specific
- Architectural patterns
  - Program organization at large
- Design patterns
  - Ways to concrete solutions, are problem-specific
- Idioms
  - Language-specific patterns that work at code level

Idioms

- Object creation
- for..in enumeration
- Fluent interfaces
- Helpers

Good Software Architecture

Clean Code

- What is clean code?
  - complex != complicated
  - Principles that help design code that is easy to
    - Test
    - Maintain
    - Extend

Principles

- KISS – keep it simple, stupid
- DRY – don‘t repeat yourself
- SoC – deparation of concerns
- SLA – single level of abstraction
- SOLID
  - SRP – single responsible principle
  - OCP – open closed principle
  - LSP – Liskov supsitution principle
  - ISP – interface segregation principle
  - DIP – dependency inversion principle

Don‘t repeat yourself (DRY)

- Reusable pieces - no copy and paste
  - structure code into classes, methods, functions

- Imposes a level of abstraction
  - Sometimes it is necessary to duplicate exact same or slightly modified code
  - Depending on programming language choice this might be more or less.

Keep it simple, stupid (KISS)

- „Everything should be made as simple as possible, …“ – Albert Einstein
- Solutions should be simple and easy to understand – even tomorrow, next year or by the new developer in the team
- „but not simpler“ – sometimes Software developers tend to be „clever“
  - Don‘t be „clever“

Single responsibility principle

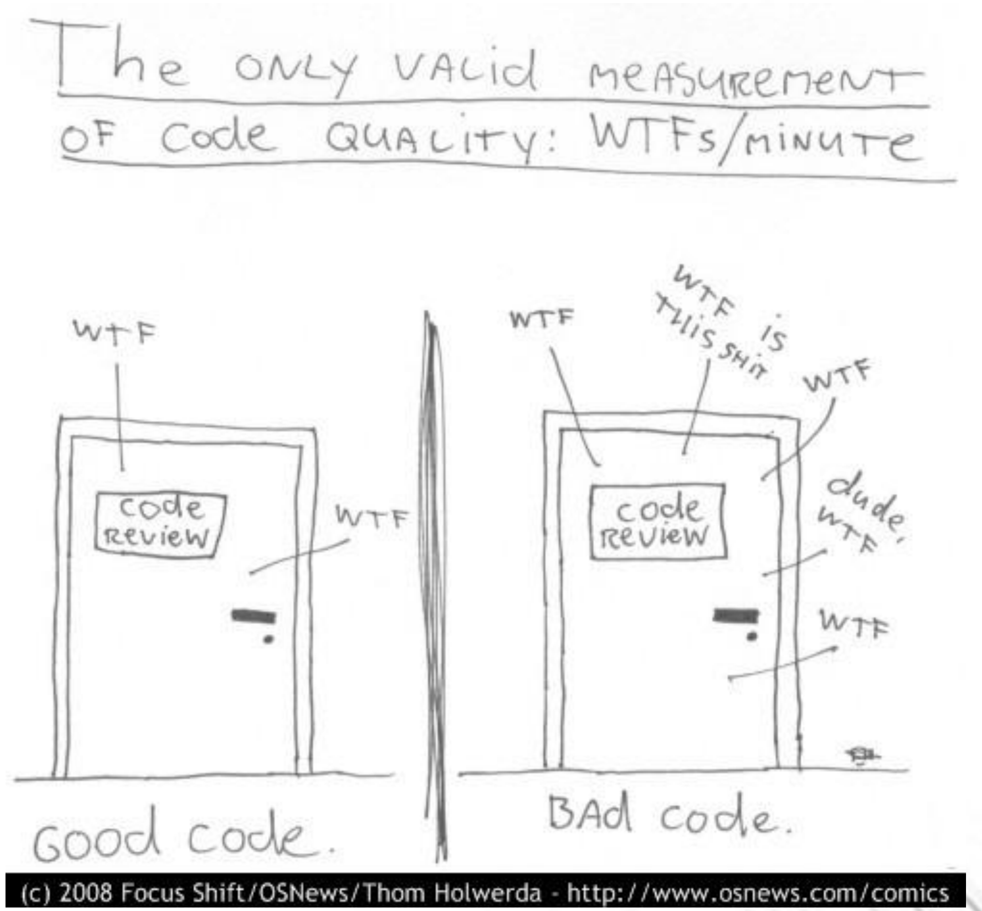
- A class should only have one responsibility
- Modifications should have as little impact as possible to other parts
- higher complexity makes it harder to understand code

Separation of concerns (SoC)

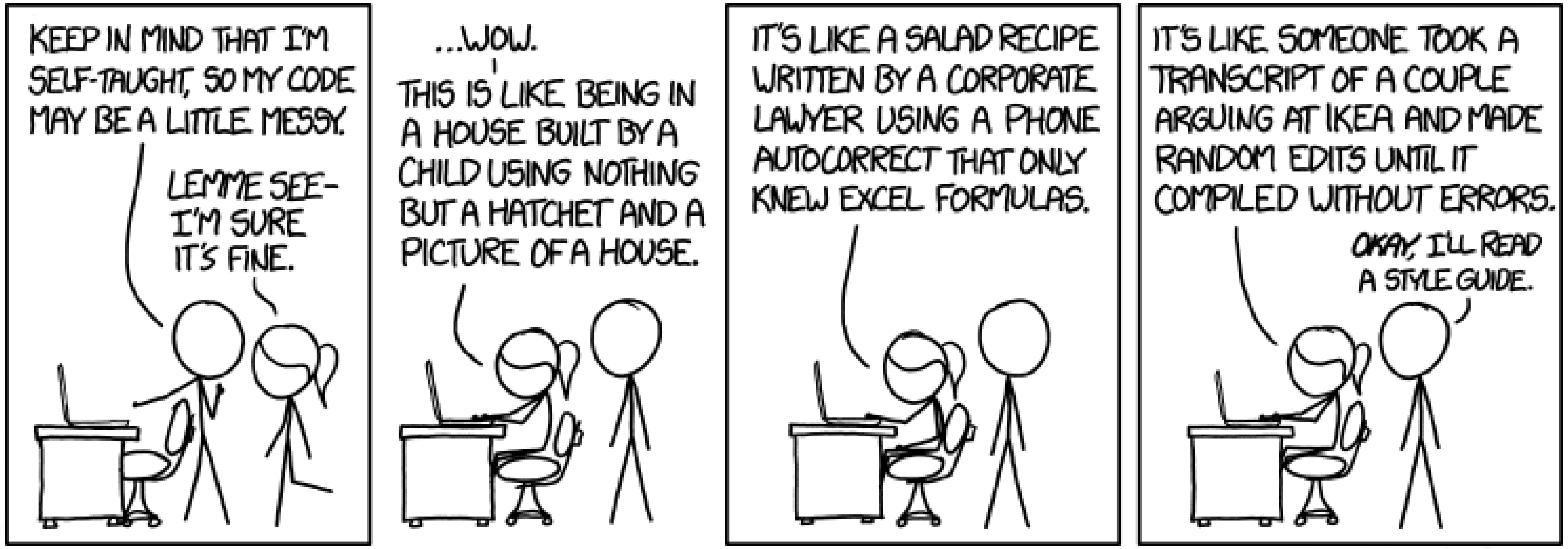
- Releated to the SRP but expands beyond a class
- Concerns are for example:
  - Logging, persistence, caching
- Depending on the programming language it might be easier
  - AOP makes separation and integration very easy

Principle of least astonishment

- Software and source code behavior should not surprise you


The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Code quality – code review


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

Algorithms

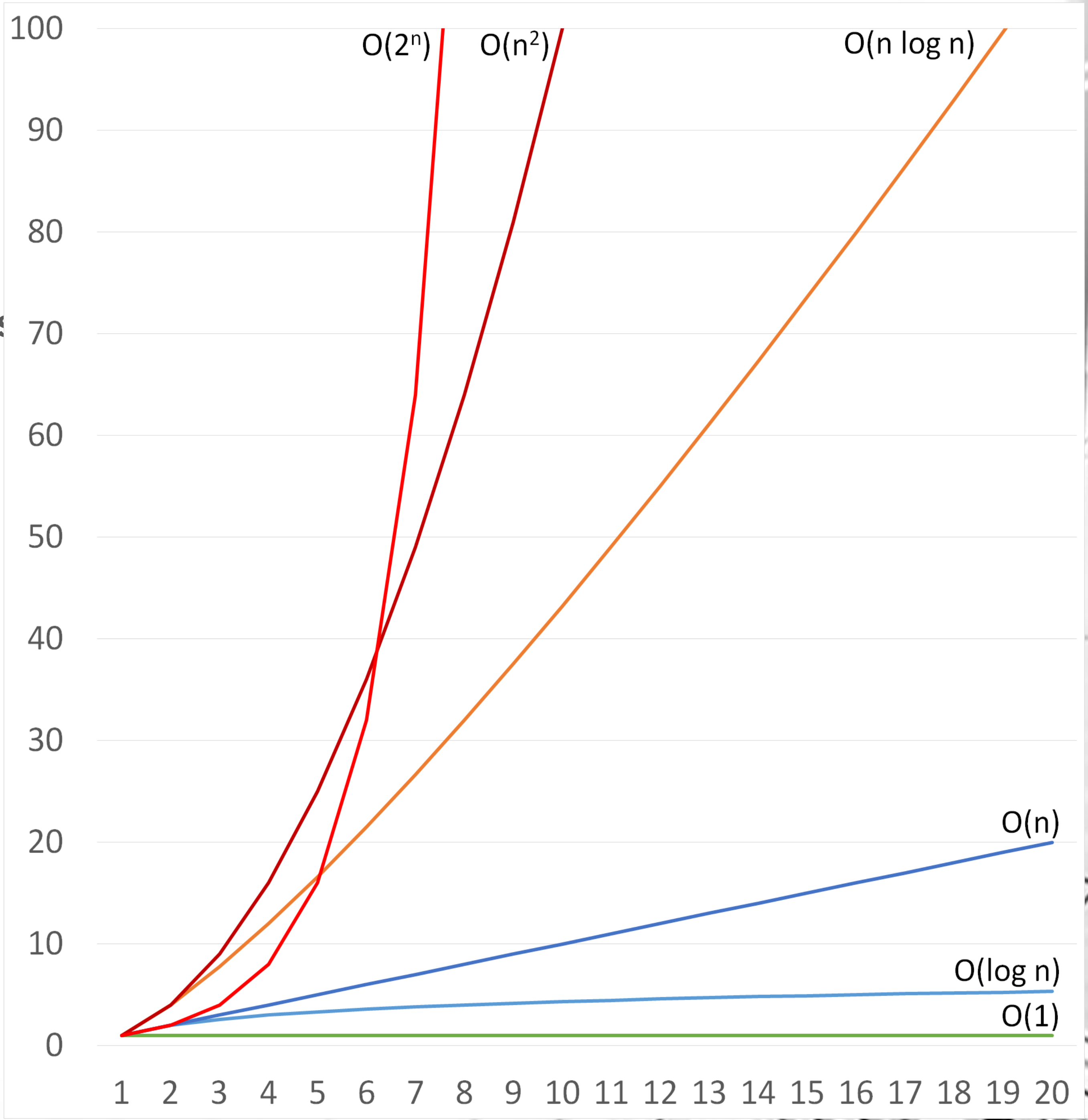
- Different algorithms scale differently
- What works in testing may fail in production

Algorithms complexity

- Tells us how algorithm slows down if data size is increased by a factor of n
- O()
  - O(n), O(n^2), O(n *log n*) …
- Time **and** space complexity

Typical complexities

• $O(1)$	accessing array elements
• $O(\log n)$	searching in ordered list
• $O(n)$	linear search
• $O(n \log n)$	quick sort (average)
• $O(n^2)$	quick sort (worst), naive sort (bubblesort, insertion, selection)
• $O(c^n)$	recursive Fibonacci, travelling salesman



Comparing complexities

Data size	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(c^n)
1	1	1	1	1	1	1
10	1	4	10	43	100	512
100	1	8	100	764	10.000	10^{29}
300	1	9	300	2.769	90.000	10^{90}

Optimization

- Jackson’s First Rule of Program Optimization:

  “Don’t do it.”

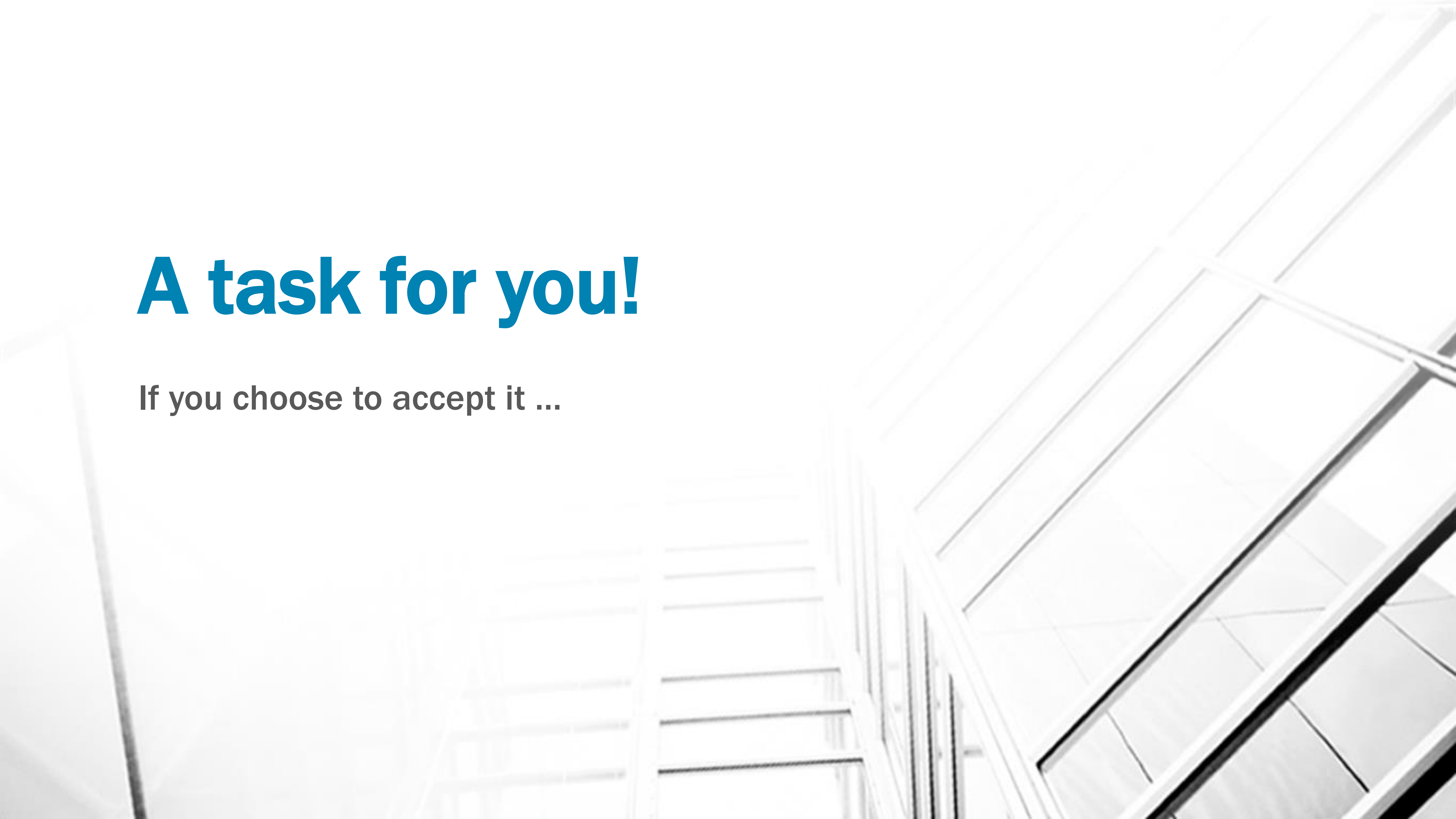
- Jackson’s Second Rule of Program Optimization (for experts only!):

  “Don’t do it yet.”

  - Michael A. Jackson

What to optimize?

- Is slow too slow?
  - Slower code may be easier to maintain
  - It is hard to optimize extremely fast code
- Measure first!
  - TStopwatch
  - Profilers
- Measure on real data
  - “Now executes empty loop 2.5 times faster!”
  - “It works fine on this table with 10 entries!”



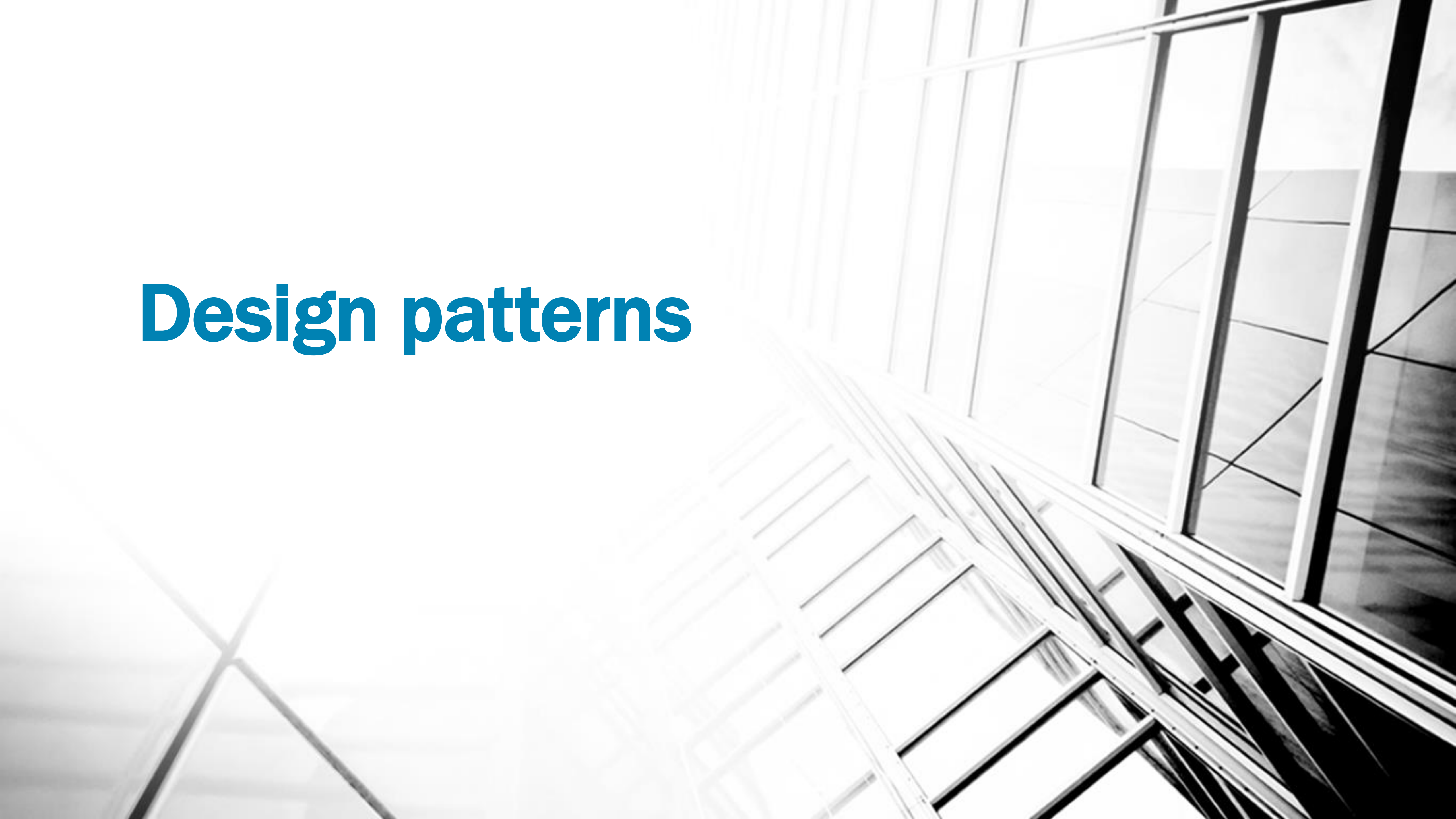
A task for you!

If you choose to accept it …

Task 2_3\Task23

Use RTL function with better O complexity to make the program run faster.

Design patterns



Design patterns

- Distillation of multiple solutions
- Language-agnostic (partially)
- Template for problem-solving
- Custom vocabulary

Design patterns are NOT

- A way to architect the program
- A silver bullet
- A detailed recipe
- A goal

“Gang of Four”

- “Design Patterns: Elements of Reusable Object-Oriented Software”, 1994

  Erich Gamma, Richard Helm, Ralph Johson, John Vlissides
- 23 patterns
- In parts very OO-specific
- Composition over inheritance

Classification

1. Creational patterns
   - Creating new objects and groups of objects
2. Structural patterns
   - Compose objects to create new functionality
3. Behavioral patterns
   - Object cooperation
4. Concurrency patterns
   - Parallel programming

Use design patterns!

- Well, use all of them except one!
- Pop quiz – which one?

A. Facade	B. Singleton
C. Observer	D. Thread pool

Examples: Singleton

- A class with only one instance
- Hard to implement correctly in Delphi
- Hard to mock
- Hard to configure
- “Don’t be an idiot” works just as good

Examples: Factory method

- Defer creation of internal data to a subclass
- Better ways to do it than with subclassing

Examples: Iterator

- Traverse a container and access the container’s elements
- Enumeration
- for..in

Examples: Observer

- Notify dependent objects on state changes
- Key part of the Model-View-Controller architectural pattern

A task for you!

If you choose to accept it ...

Task 2_4\Task24

Introduce a pattern to make the code easier to maintain.

Parallel Programming

Parallel programming

“Don’t do it.”

“Don’t do it yet.”

Unit testing

- Very hard to do!
  - Hard to test interactions between threads
  - Hard to test edge cases
  - Hardest of all: **anticipate** problems
- Also very important!

Clean code

- Immutable data structures
- Pure methods (no side effects)

Architecture

- Messaging instead of locking
  - Not always possible!
  - Proper locking!
  - Deadlocks

Architecture

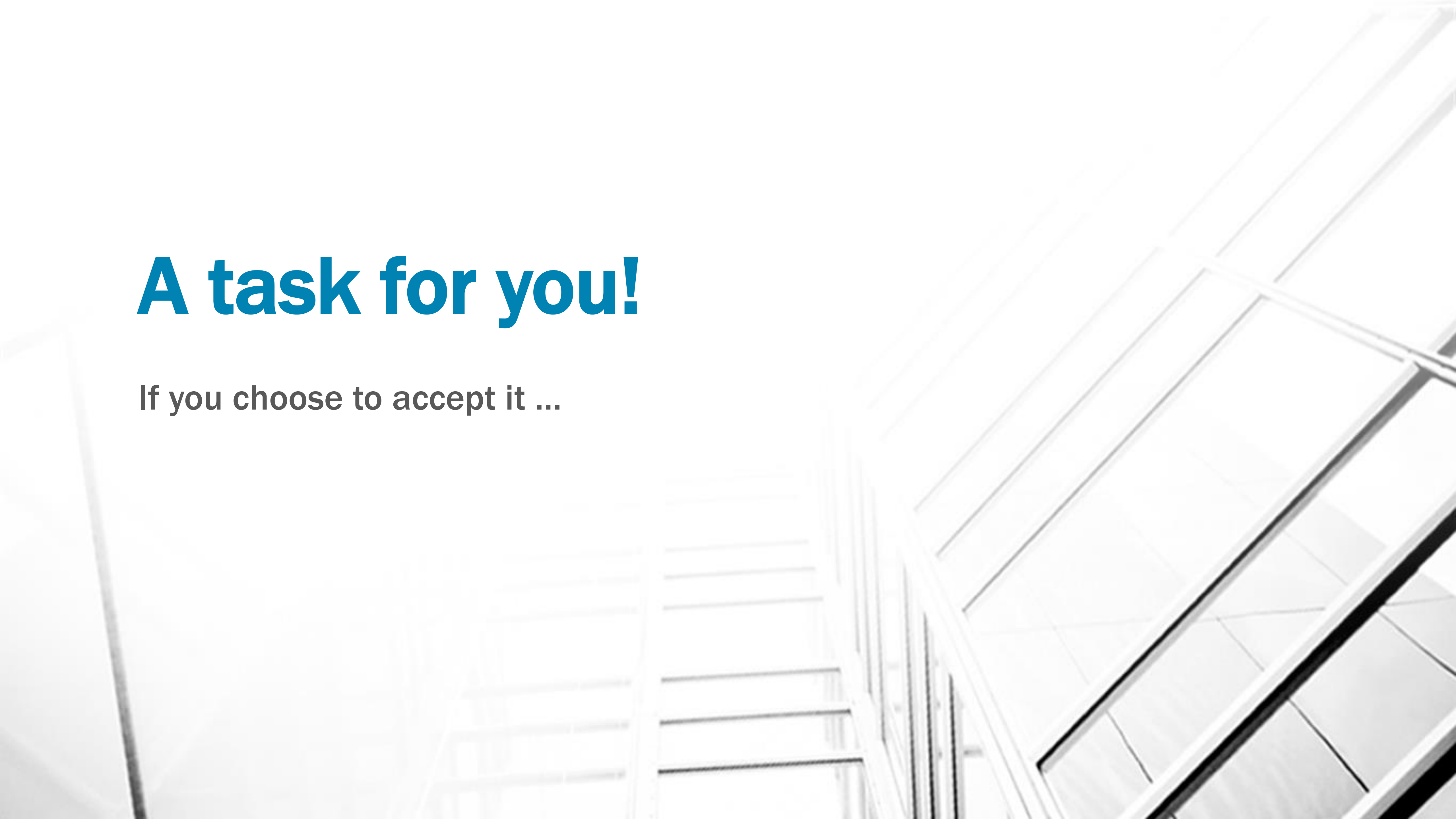
Don’t access GUI from a background thread!

No globals!

- Shared state introduces problems
- Not always possible to remove => locking

Patterns

- Locking
  - Lock & test
  - Test & lost & test again
- Atomic modifications
- Messaging
- Pipelining



A task for you!

If you choose to accept it ...

Task 2_5\Task25

Fix the program so that it will work correctly.