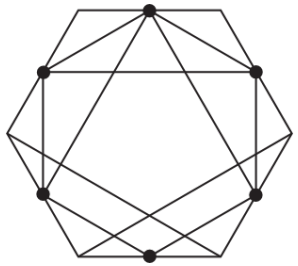
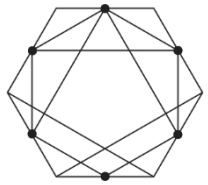


Clean Code
Functions

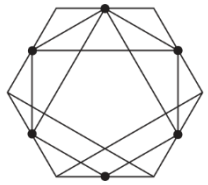


FIRST RULE ABOUT FUNCTIONS

Functions should be small.

Even smaller than that!

- **No more than 4 lines!**

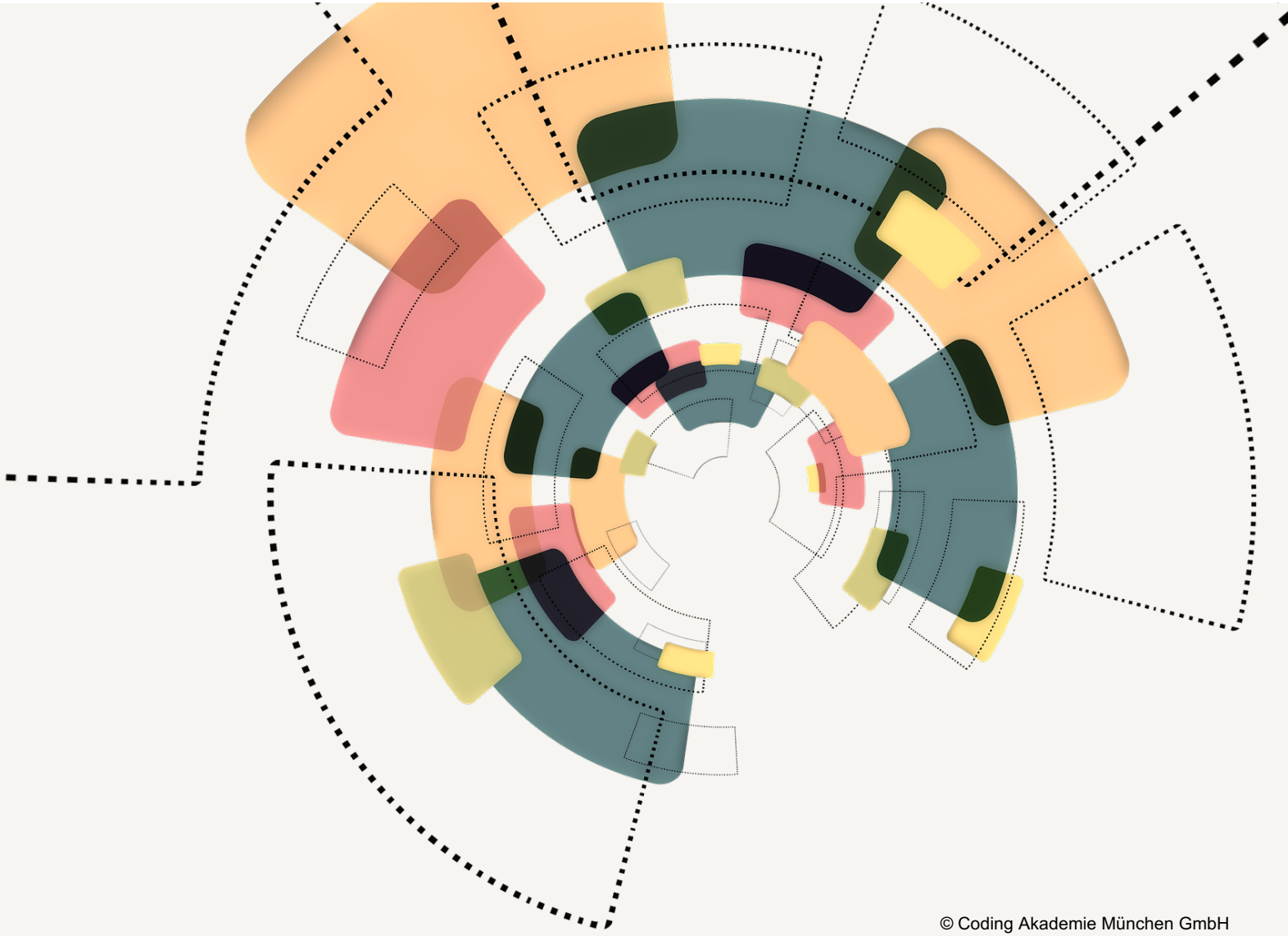


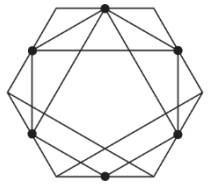
DO ONE THING

- Functions should do one thing
- They should do it well
- They should do it only

ABSTRACTION LEVELS

Everything the function does in its body should be one (and only one) level of abstraction below the function itself.





“TO” PARAGRAPHS: CHECKING LEVELS OF ABSTRACTION

TO RenderPageWithSetupsAndTeardowns

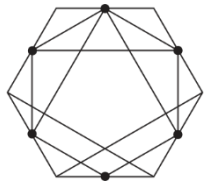
We check to see whether the page is a test page and

if so, we include the setups and teardowns.

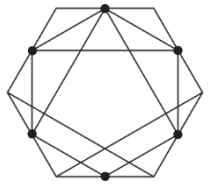
In either case we render the page in HTML

THE STEPDOWN RULE

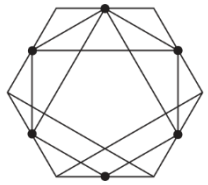
- We want code to read like a top-down narrative
- Every function should be followed by those one level of abstraction below it



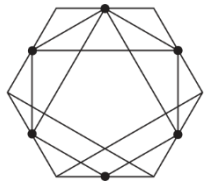
```
Money calculatePay(Employee e) {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
}
```



```
Struct EmployeeFactory {
    std::unique_ptr<Employee> makeEmployee(EmployeeRecord r) {
        switch (r.type) {
          case COMMISSIONED:
            return std::make_unique<CommissionedEmployee>(r);
          case HOURLY:
            return std::make_unique<HourlyEmployee>(r);
          case SALARIED:
            return std::make_unique<SalariedEmploye>(r);
          default:
            throw InvalidEmployeeType(r.type);
        }
    }
}
```

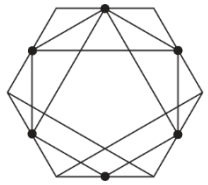


```
Money CommissionedEmployee::calculatePay(Employee e) {
    return calculateCommissionedPay(e);
}

Money HourlyEmployee::calculatePay(Employee e) {
    return calculateHourlyPay(e);
}

Money SalariedEmploye::calculatePay(Employee e) {
    return calculateSalariedPay(e);
}
```



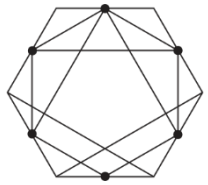
MORE RULES FOR FUNCTIONS

- Use descriptive names
- Use few (or no) arguments
- Don’t use Boolean arguments (flag arguments)
- Should have no hidden side effects



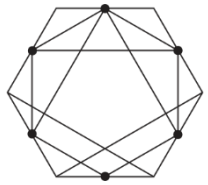
HIDDEN SIDE EFFECTS

```
bool checkPassword(std::string userName, std::string password) {
    User& user = UserGateway.findByName(userName);
    if (user != User.NULL) {
        std::string codedPhrase = user.getPhraseEncodedByPassword();
        std::string phrase = cryptographer.decrypt(codedPhrase, password);
        if (phrase == "Valid Password") {
            session.initialize();
            return true;
        }
    }
    return false;
}
```



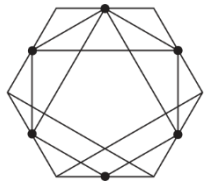
AVOID OUTPUT ARGUMENTS

```
HitResult hit_result;
player.check_collision(obstacle, hit_result);
if (hit_result.collision_occurred) {
    …
}
```



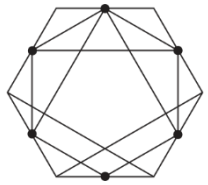
AVOID OUTPUT ARGUMENTS

```
void check_collision(const Actor& a, HitResult& hit_result);
```



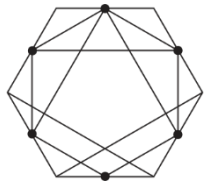
AVOID OUTPUT ARGUMENTS

```
std::unique_ptr<HitResult> check_collision(const Actor& a);
```



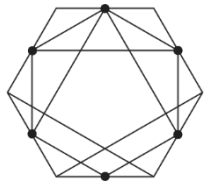
AVOID OUTPUT ARGUMENTS

```
auto hit_result{player.check_collision(obstacle)};


if (hit_result.collision_occurred) {
    …
}
```



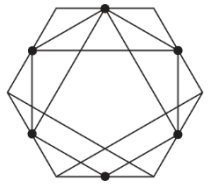
COMMAND QUERY SEPARATION

```
bool has_default_value() {
    if (default_value >= 0)
        return true;
    else
        default_value = 123;
        return false;
```

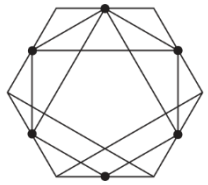


COMMAND QUERY SEPARATION

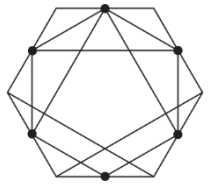
```
bool has_default_value() {
    return default_value >= 0;
}
void set_default_value() {
    default_value = 123;
}
```



```
ErrorCode print_result() {
   if (!outstream) {
      return ErrorCode::STREAM_UNAVAILABLE;
   }
   …
   return ErrorCode::NO_ERROR;
}
```



```
void print_result() {
    if (!outstream) {
        throw std::runtime_error("No stream available");
    }

    …
}
```



DRY: DON’T REPEAT YOURSELF

- Try to eliminate duplicated code
  - It bloats the code
  - It requires multiple modifications for every change
- But: often duplicated code is interspersed with other code
- Take into account the scope in which you keep code DRY!