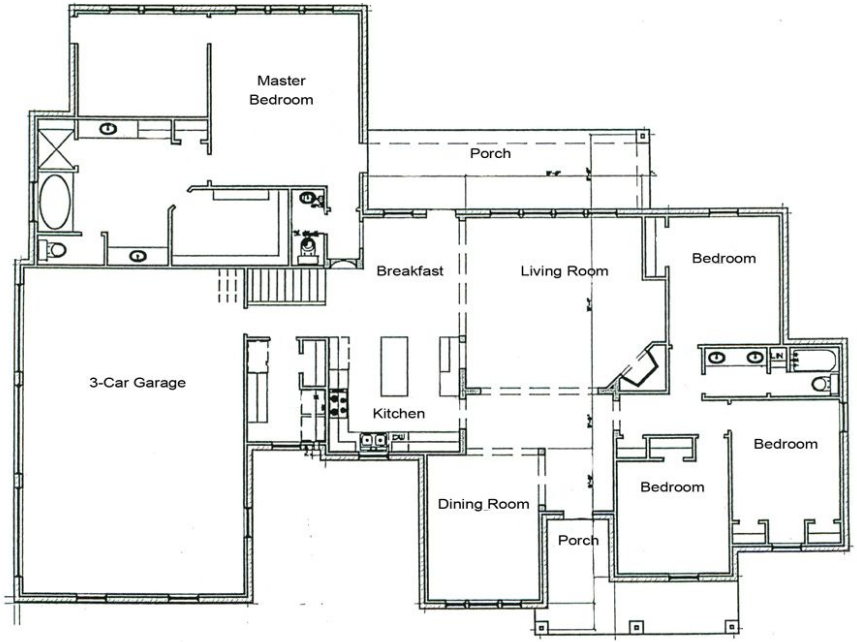
Agenda

- What is architecture
- The benefits of architecture
- How to create architecture

What is architecture

- The shape of the system
- Involves all levels



The benefits of architecture

- Behaviour and structure
- Reduce amount of work and improve efficiency
- Support development, deployment, maintenance

Development

- A system that is difficult to develop has a short lifespan
- Small teams can do without good architecture
- Large systems need good architecture

Deployment

- Deployment should be as easy as possible
- Can be difficult with a microservice architecture

Maintenance

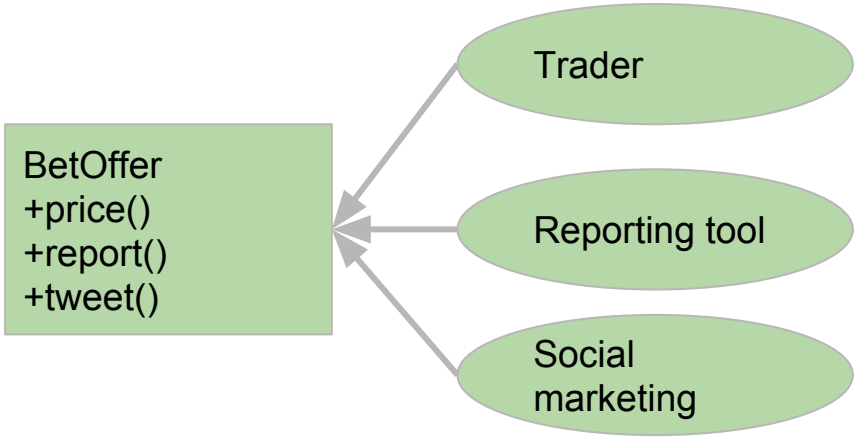
- Should be easy to find where to make changes
- And to make them safely

How to create architecture

- SOLID principles
- Component principles
- Independence
- The clean architecture

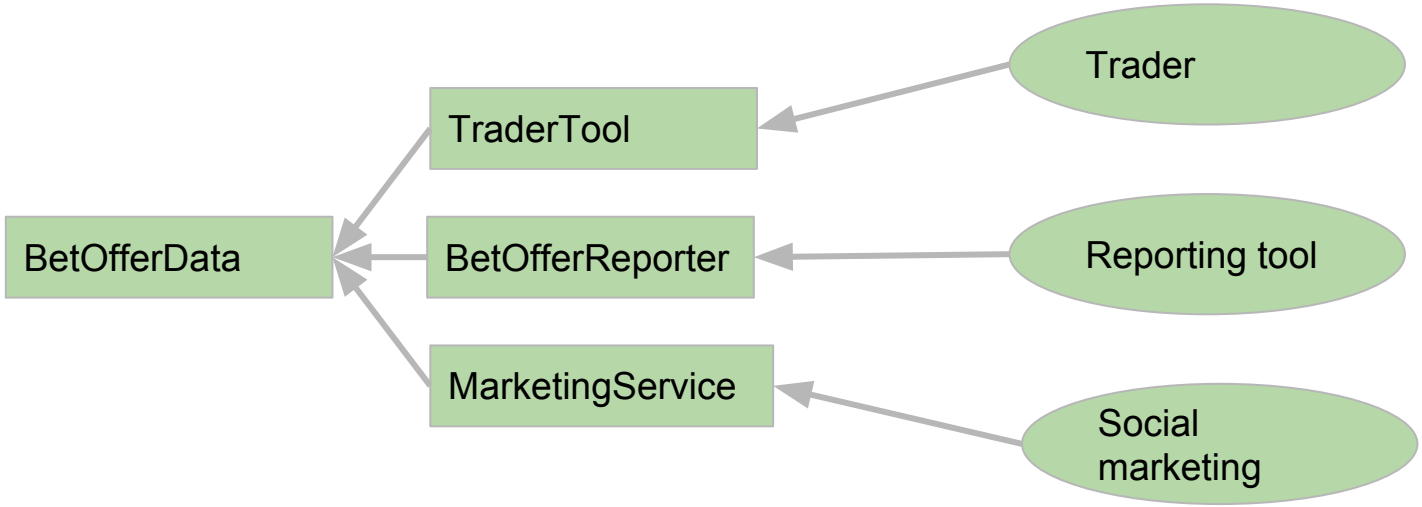
Single responsibility principle

- A module should only have one reason to change



Single responsibility principle

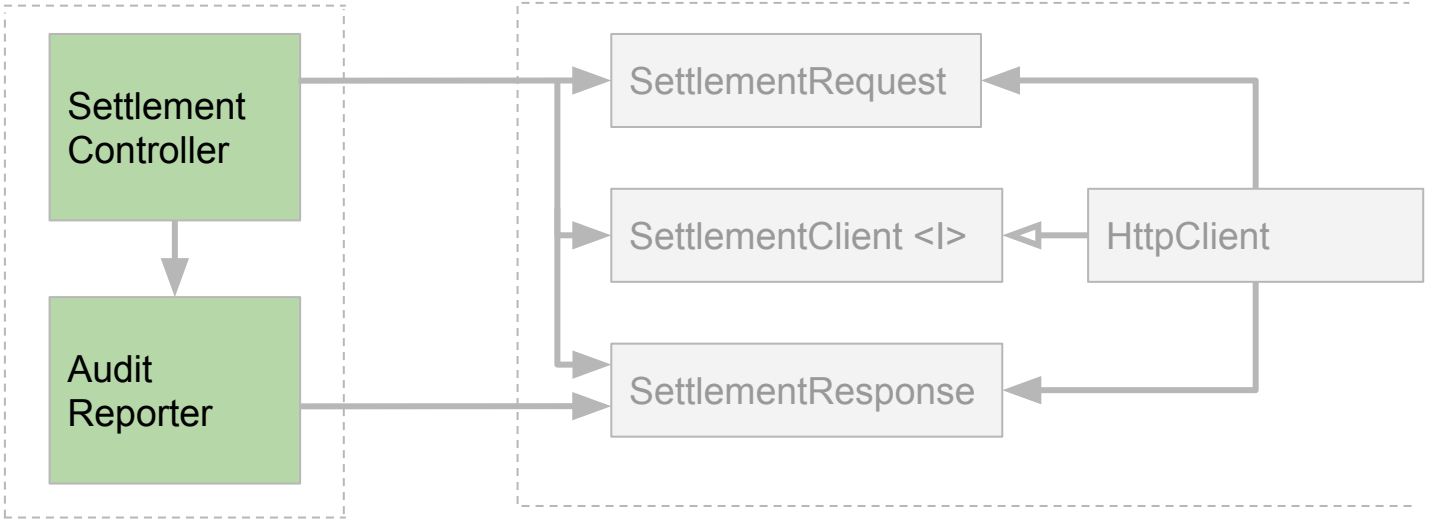
- A module should only have one reason to change
- Can be solved by separating data and functionality



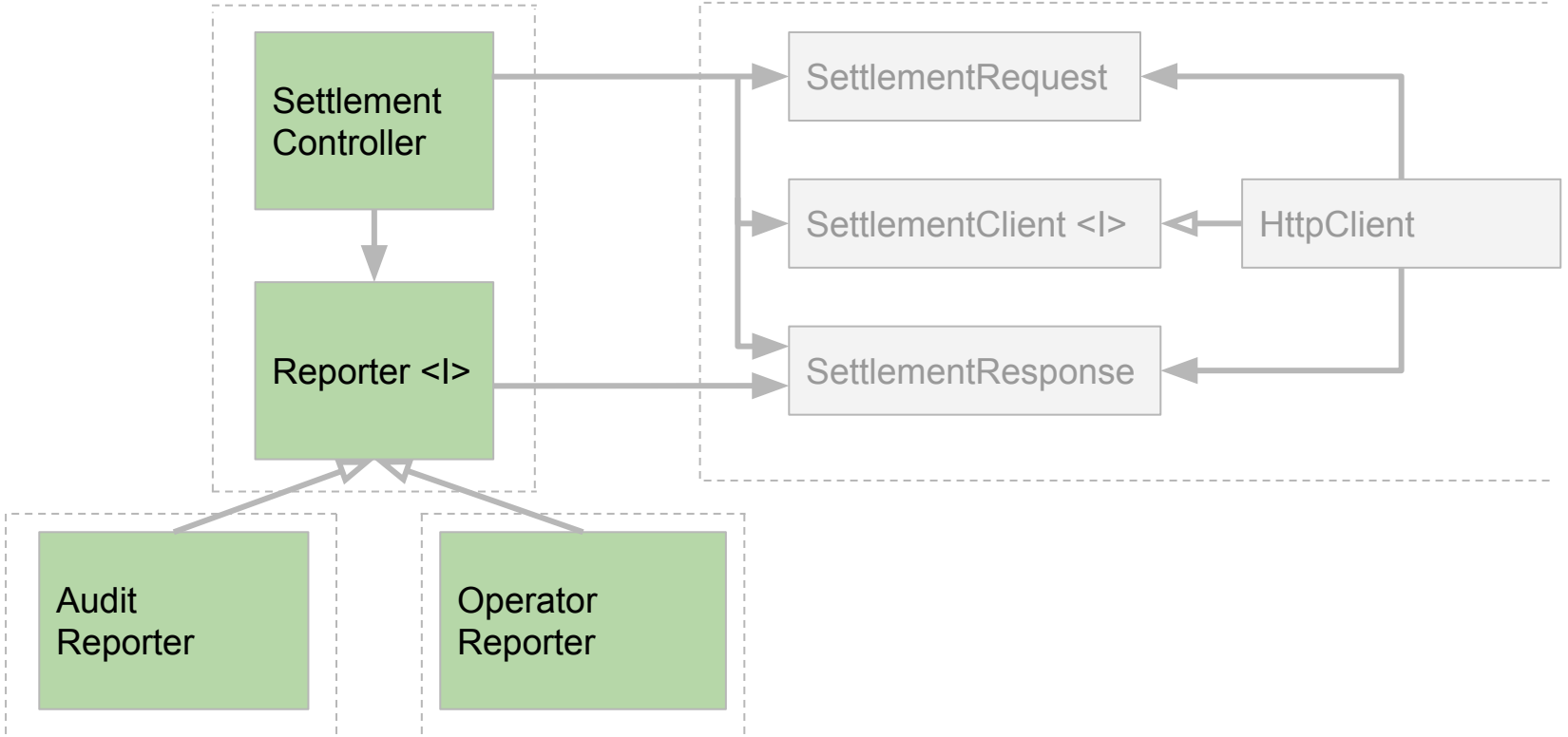
Open closed principle

- Open for extension, closed for modification
- Requirement changes should not necessitate code changes

Example: Reporting settlement information

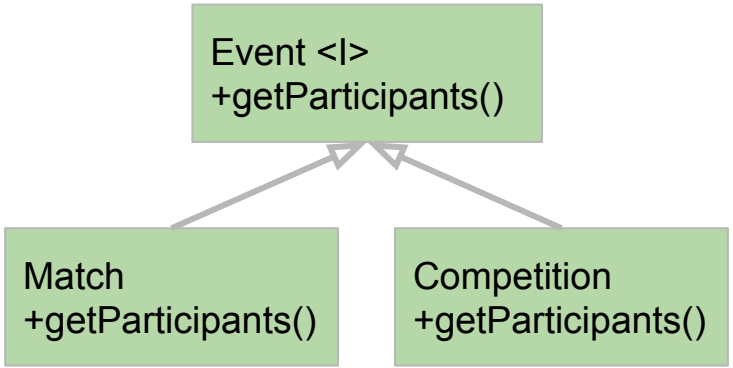


Example: Reporting settlement information



Liskov substitution principle

- Subtypes should be substitutable

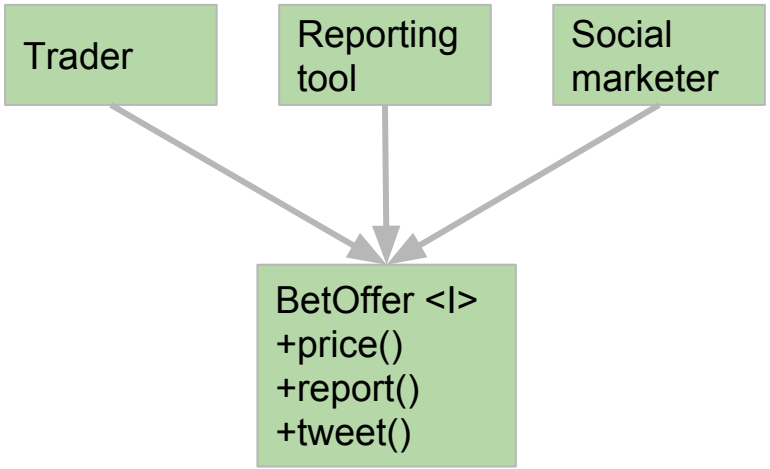


Example: LSP violation

- A service that reports payouts for customers' punters: /punter/123/payout
- But one customer use a different path: /payout/punter/123
- Violation leads to a lot of more work if we want a clean solution

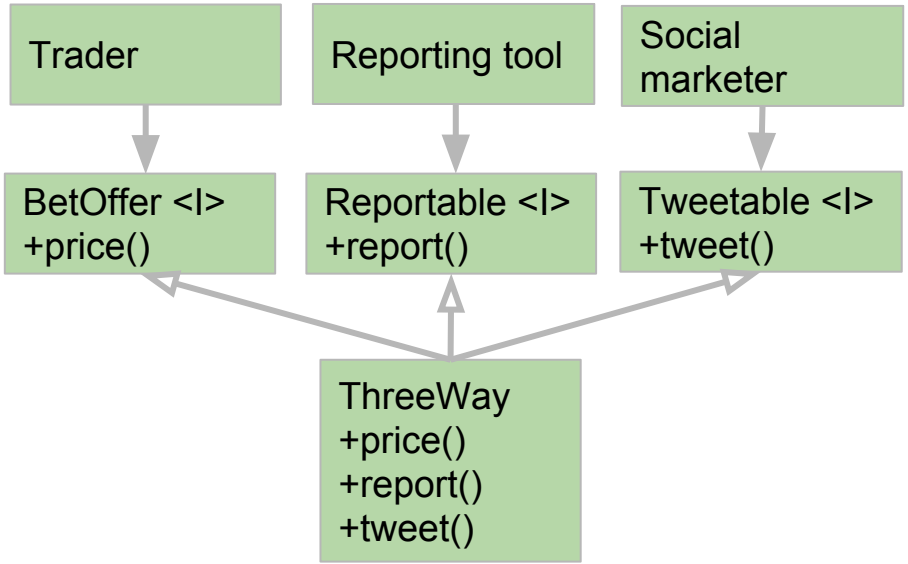
Interface segregation principle

- Large interfaces forces users to depend in things they don’t need



Interface segregation principle

- Large interfaces forces users to depend in things they don’t need



Interface segregation principle

- Large interfaces forces users to depend in things they don’t need

```
public void settleThreeWay(final int winningOutcome) {
        settlementSender.send(new Settlement() {
                @Override public int getWinningOutcome() { return winningOutcome; }
        });
}
```

Interface segregation principle

- Large interfaces forces users to depend in things they don't need

```
public void settleThreeWay(final int winningOutcome) {
    settlementSender.send(new Settlement() {
        @Override public int getWinningOutcome() { return winningOutcome; }
        @Override public int getCompetitionId() { return -1; } // not a competition
        @Override public String getReport() { return "Default report"; }
        @Override public String getTweet() { return "I'm a settlement!"; }
        @Override public void onSettled() { /* don't care */ }
    });
}
```

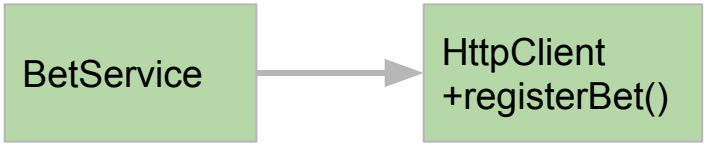
Dependency inversion principle

- Want to separate business logic from database, UI, etc.
- Higher levels need to call lower levels
- Invert the dependency using an interface

Dependency inversion principle

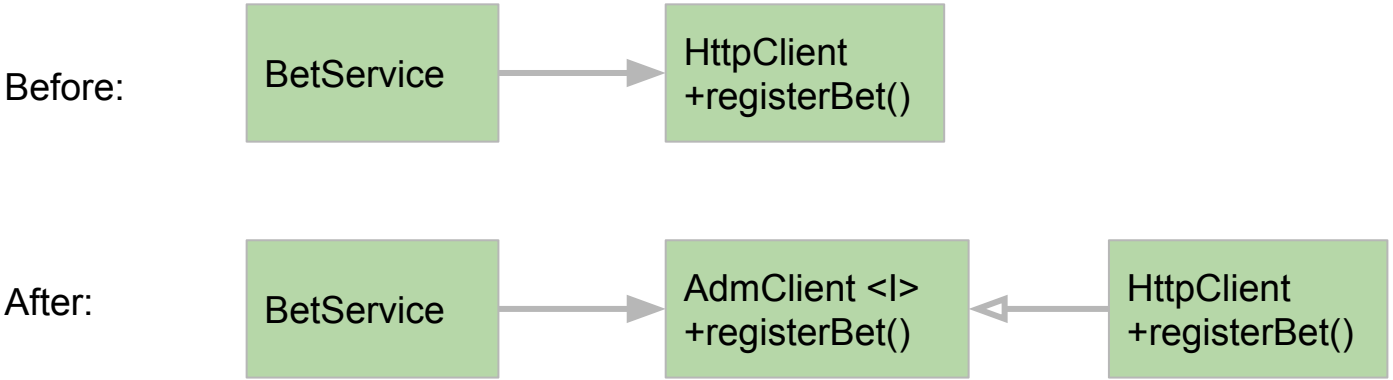
- Want to separate business logic from database, UI, etc.
- Higher levels need to call lower levels
- Invert the dependency using an interface

Before:



Dependency inversion principle

- Want to separate business logic from database, UI, etc.
- Higher levels need to call lower levels
- Invert the dependency using an interface



Component principles

- SOLID for components
- Component cohesion
- Component coupling

Module - a set of related functionality, a class, a package, a jar

Component - the smallest entity that can be deployed, a jar

Component cohesion

- Reuse/release equivalence principle
- Common closure principle
- Common reuse principle

Reuse/release equivalence principle

- Modules in a component should make sense to be released together
  - They have a common theme
  - They should be reusable
- Kambi-commons
  - The modules have something to do with Kambi
  - Should not include NiroTwitterClient because it’s not reusable

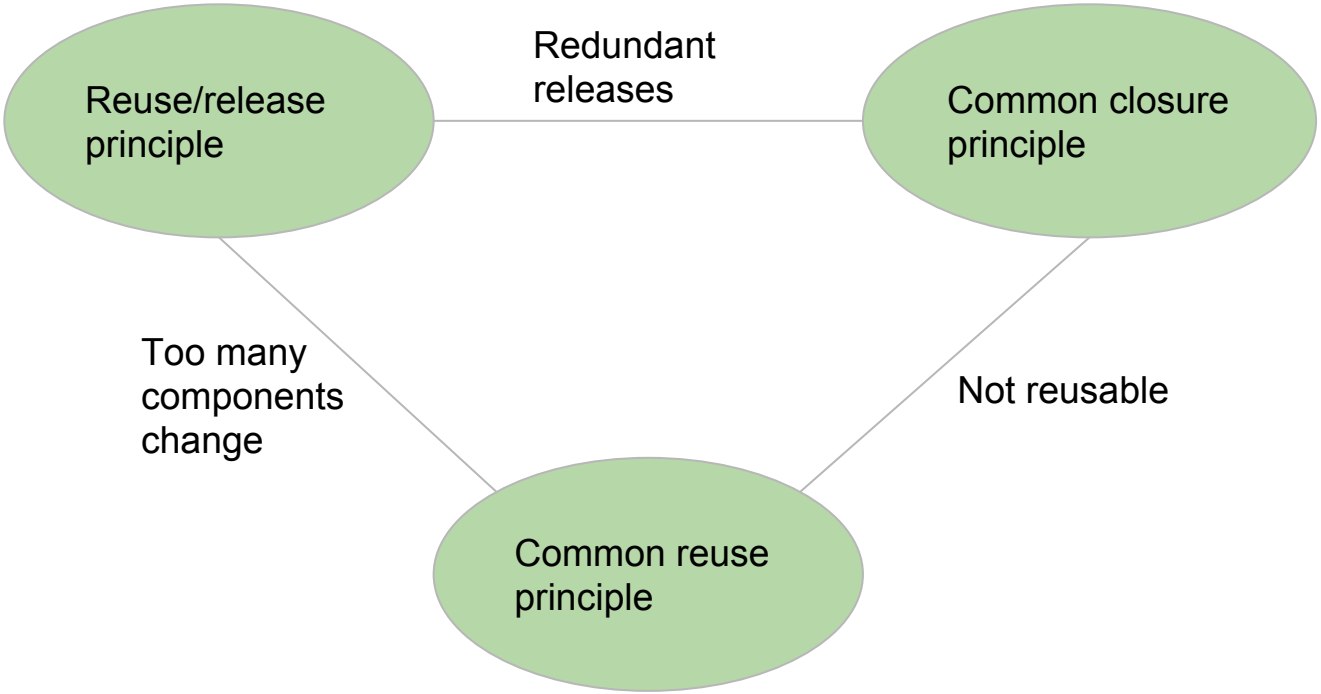
Common closure principle

- Gather modules which change at the same time and for the same reasons
- When requirements change, changes are restricted to a single component
- Kambi-commons
  - CustomerEnum is changed for the same reason and rate as ExtApiEndpointContext
  - CustomerEnum is changed for a different reason and rate than TennisSet

Common reuse principle

- Don’t force the user to depend on things it doesn’t need
- A user should need to depend on all modules in a component
- Kambi-commons
  - If you need monitoring, you’ll probably use all classes in the monitoring package
  - But you might not want to be forced to depend on TwitterMonitorAdapter

Component cohesion

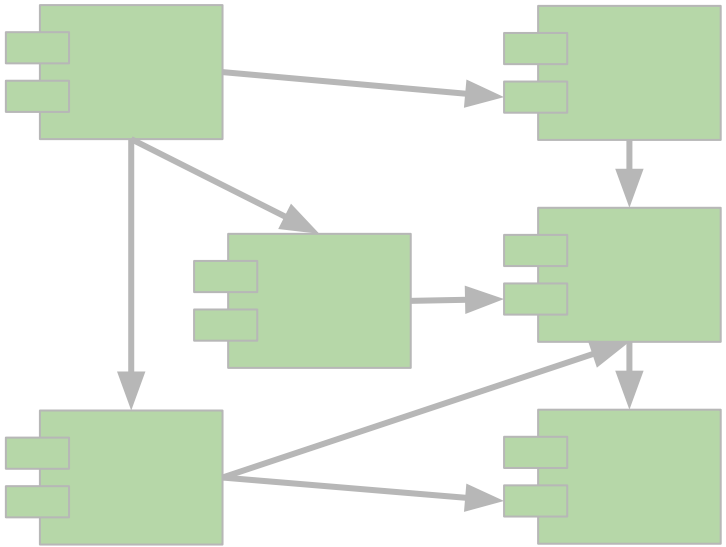


Component coupling

- Acyclic dependencies principle
- Stable dependencies principle
- Stable abstractions principle

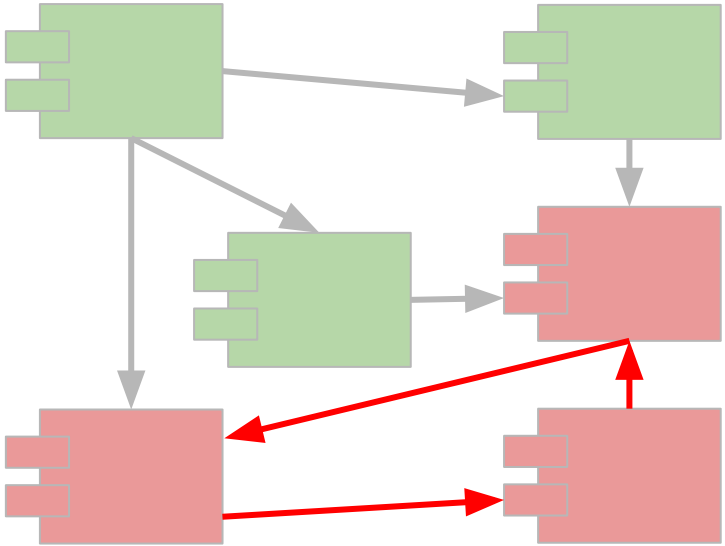
Acyclic dependencies principle

- Don’t want dependency cycles



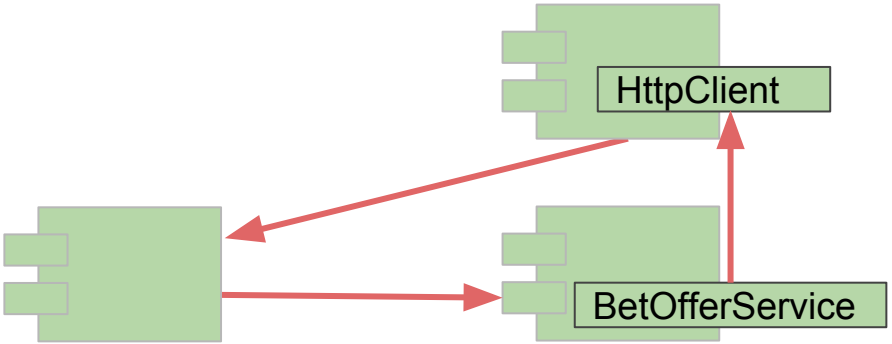
Acyclic dependencies principle

- Don’t want dependency cycles
- If a cycle is formed, we’ve got trouble



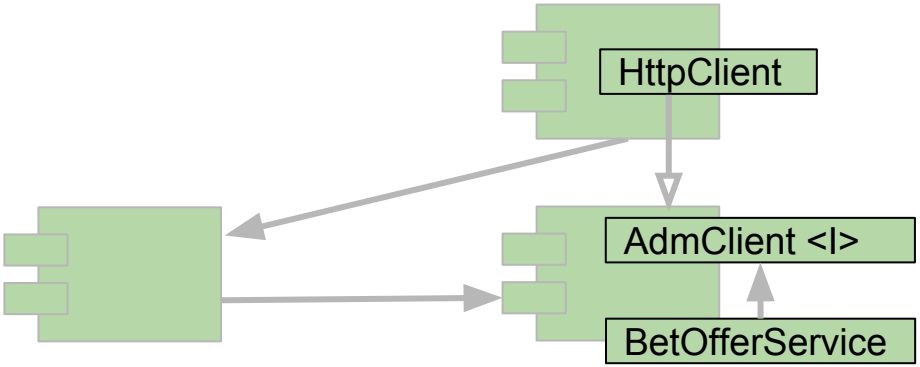
Acyclic dependencies principle

- Cycles can be broken using Dependency inversion principle



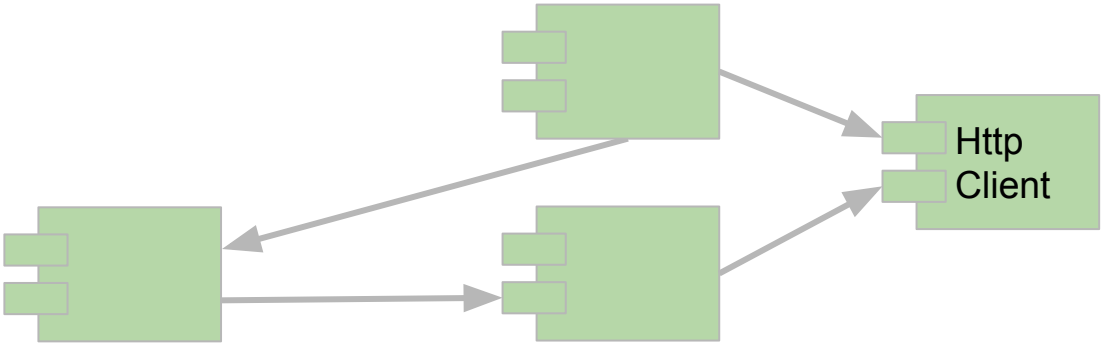
Acyclic dependencies principle

- User depends on an abstraction



Acyclic dependencies principle

- Can extract the dependency into a new component

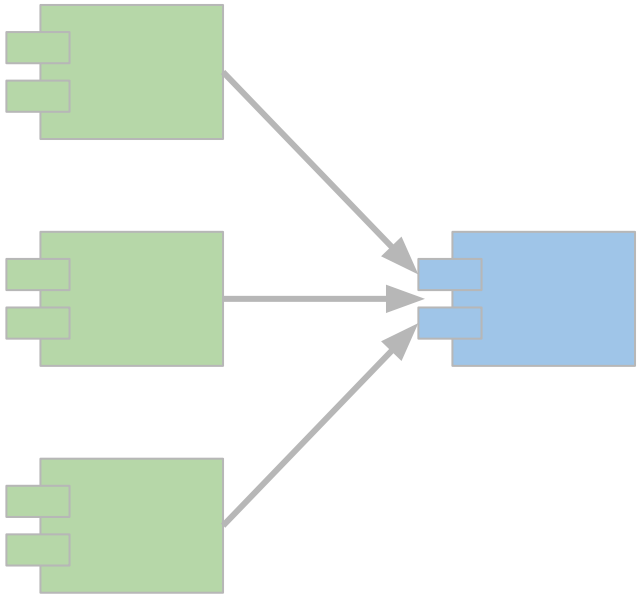


Stable dependencies principle

- Always depend on things that are more stable than yourself

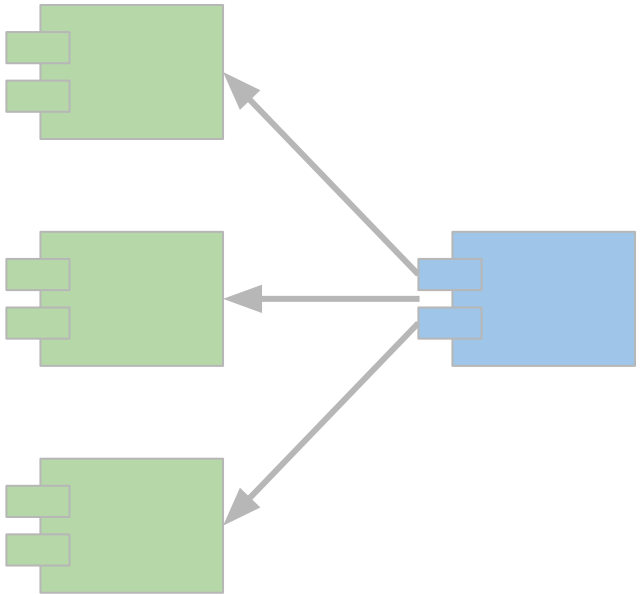
Defining stability

- Something is stable if many things depend on it



Defining stability

- Something is unstable if it depends on many things

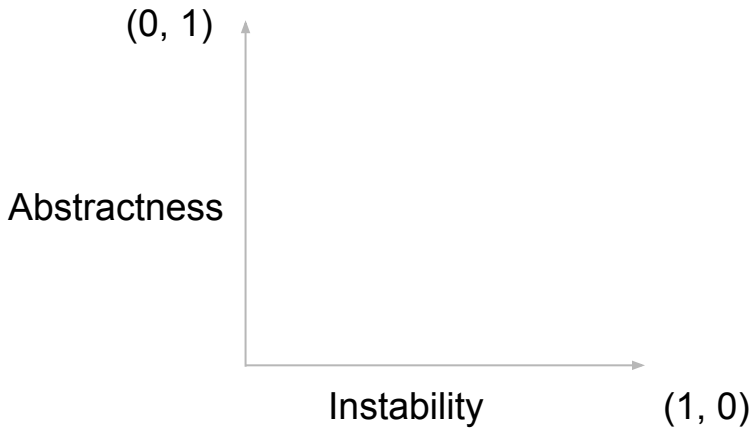


Stable abstractions principle

- A component should be as abstract as it is stable
- How can a stable component be flexible enough to handle changes?
- SDP says that dependencies should be in the direction of stability
- SAP says that stability is abstraction
- So dependencies should point to abstractions

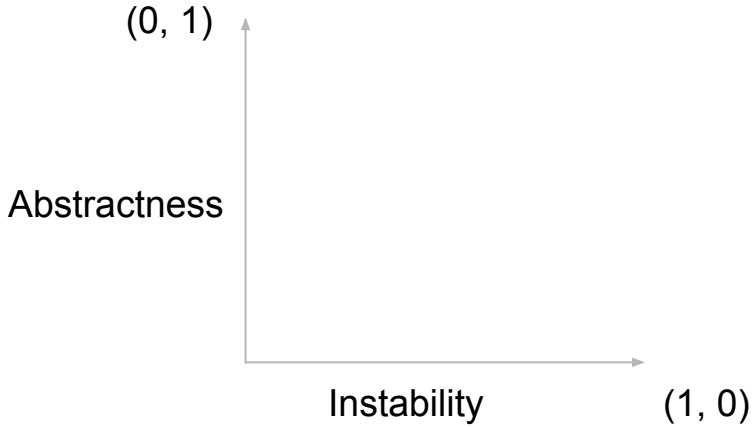
Abstractness vs. Instability

- Abstractness = (# abstract classes) / (# classes)
- Instability = Fan out / (Fan out + Fan in)



Abstractness vs. Instability

- (0, 0) is difficult to extend and modify
- (1, 1) is useless
- Want components that are pure: (0, 1) and (1, 0)



Independence

- Use cases
- ~~Policies~~
- ~~Levels~~

Use cases

- A use case is a description of how the system is used
- To support use cases is the architecture's highest priority
- If it’s a shopping cart, the architecture should show that
- Finding each use case should be easy

Use cases cont.

- Use cases may change for various reasons
- They are small vertical slices in the system
- Want to be able to add use cases without affecting existing ones
- Allow for scaling specific use cases (microservices)

The clean architecture

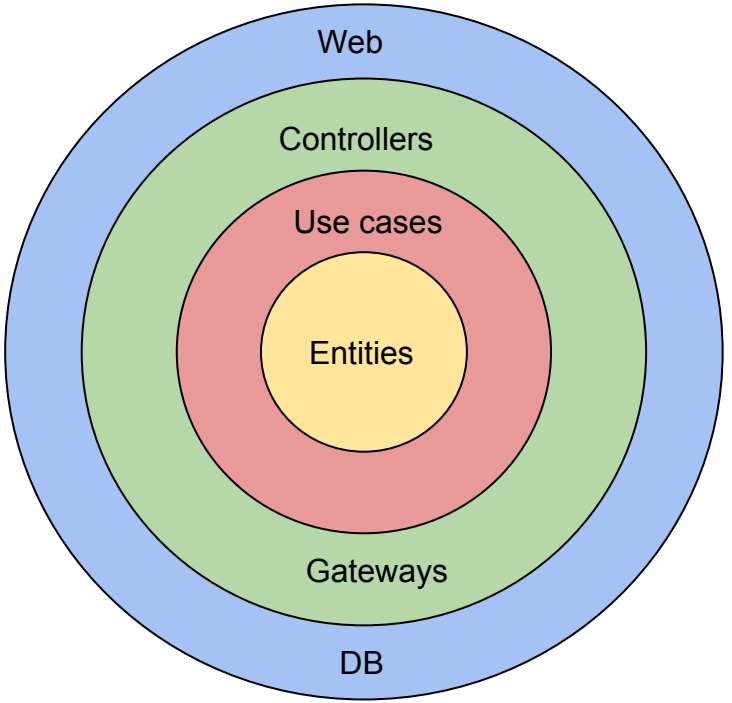
- Common architectures
  - Ports and adapters / Hexagonal
  - Data, Context, Interaction
  - Boundary, Control, Entity
- Focused on separation of concerns
- (Screaming architecture)

Screaming architecture

- Is the architecture screaming health care system or Spring MVC?
- The architecture supports the system’s use cases, not frameworks
- A good architecture should let all use cases be testable without any frameworks

The clean architecture

- The source code dependencies point towards the middle
- An inner circle does not know about the outer circle
- Entities embodies the critical business rules
- Use cases are the application specific rules
- Frameworks are where are the details are (DB, Web, etc.)
- Can of course use more than four circles



End

- Not covered
  - Policies
  - Levels
  - Humble objects
  - Boundaries
  - Services
  - Packaging strategies