Object-Oriented Design

Georgia Tech

Software Design

Georgia Tech

Design

Design (noun) A plan or protocol for carrying out or accomplishing something. – Webster's Dictionary

Engineering design: A systematic, intelligent process in which designers generate, evaluate and specify designs for devices, systems, or processes whose form(s) and function(s) acheive clients' objectives and users' needs while satisfying a specified set of constraints. – Dym and Little, quoted in Carlos Otero, Software Engineering Design

Design Taking choices among alternative solutions in pursuit of design goals and satisfying design constraints. – Dr. CS

Georgia Tech

Fundamenta Design Principles

- Modularization
- Abstraction
- Encapsulation
- **Coupling and Cohesion**
- Separation of interface and implementation
- Suficiency and completeness

Georgia Tech

Software Development is not Engineering – Reeves

Software development is not engineering (at least not in the traditional sense). In traditional engineering:

- Engineers (design team) create detailed designs
- Designs are given to manufacturing teams to build
- If design is complete, no further input from design team is necessary
- Building can be very expensive
- Bugs in hardware design even more expensive: e.g., if design error in a car is not caught early, thousands of cars can be sold with defects resulting in deaths and recalls, bridges can collapse, etc.

Feedback cycle very long and expensive, leading to traditional engineering's emphasis on detailed documentation

Georgia Tech

The Code is the Design – Reeves

In software development:

- Source code is given to a compiler and linker, which builds the runnable software very quickly and inexpensively
- Tight feedback loop
- Easy to generate complex designs
- Testing and debugging is part of the design process

Software design is about managing complexity.

Georgia Tech

Object-Oriented Design

Georgia Tech

Object Design – Wirfs-Brock

- Design driven by roles and responsibilities
- Write story about system, identify themes and abstractions, identify candidate roles/classes that support themes
- When candidates identified, model reponsibilities and collaborations
- Exploratory design with CRC cards (candidates, responsibilities, collaborations)
- Object roles often fit into these steriotypes:
  - Informatoin holder – knows and provides information
  - Structurer – maintains relationships between objects and information about those relationships
  - Service provider – performs work and, in general, offers computing services
  - Coordinator – reacts to events by delegating tasks to others
  - Controller – makes decisions and closely directs others' actions
  - Interfacer – transforms information and requests between idstinct parts of the system

Georgia Tech

Agile Design

Fundamental principle of agile design:

The code is the design. – Jack Reeves, 1992

Design Smells

- Rigidity – system is too hard to change becuase change in one place forces changes in many other places
- Fragility – changes break things that are conceptually unrelated
- Immobility – too hard to resuse components in other systems
- Viscosity – hard to do it right, easy to do it wrong
- Needless Complexity – infrastructure with no direct benefit
- Needless Repetition – repeated structures that could be unified under a single abstraction
- Opacity – hard to read and understand

Design smells avoided or fixed by applying design principles like SRP, OCP ...

Georgia Tech

SOLID Design – Robert C. Martin, aka, "Uncle Bob"

- **S**ingle Responsibility Principle (SRP)
- **O**pen Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

These all boil down to (high) cohesion, (loose) coupling, and reuse.

Georgia Tech

SRP Counterexample – Too Many Responsibilities

```
1  public class GreetingFrame extends JFrame implements ActionListener {
2      private JLabel greetingLabel;
3      public GreetingFrame() {
4          ...
5          JButton button = new JButton("Greet!");
6          button.addActionListener(this);
7          ...
8      }
9      public void actionPerformed(ActionEvent e) {
10         Greeter greeter = new Greeter("bob");
11         String greeting = greeter.greet();
12         greetingLabel.setText(greeting);
13     }
14 }
```

- If we add other buttons or menu items to the GUI, we have to modify the `actionPerformed` method to handle an additional event source.
- If we change the behavior of the a button, we have to modify the `actionPerformed` method.

Georgia Tech

SRP Refactoring

```
1  private class GreetButtonListener implements ActionListener {
2
3    private JLabel greetingLabel;
4
5    public GreetButtonListener(JLabel greetingLabel) {
6      this.greetingLabel = greetingLabel;
7    }
8    public void actionPerformed(ActionEvent e) {
9  ... }
10 }
```

```
1  public class GreetingFrame extends JFrame {
2    ...
3    public GreetingFrame() {
4      ...
5      button.addActionListener(new GreetButtonListener(greetingLabel));
6      ...
7    }
8  }
```

- Additions to the UI require changes only to `GreetingFrame`.
- Changes to greet button behavior require changes only to `GreetButtonListener`.

Georgia Tech

Open-Closed Principle

> *Software Entities (classes, modules, functions) should be open for extension, but closed for modification.*

- Open for extension means the module should be extendable with new behavior.
- Closed for modification means the module shouldn't need to be touched in order to add the extension.

Object-oriented polymorphism makes this possible, namely, to write new code that works with old code without having to touch the old code.

Georgia Tech

OCP Counterexample – Extension Requires Modification

```
1  public class Sql {
2    public Sql(String table, Column[] columns)
3    public String create()
4    public String insert(Object[] fields)
5    public String selectAll()
6    public String findByKey(String keyColumn, String keyValue)
7    public String select(Column column, String pattern)
8    public String select(Criteria criteria)
9    public String preparedInsert()
10   private String columnList(Column[] columns)
11   private String valuesList(Object[] fields, final Column[] columns)
12     private String selectWithCriteria(String criteria)
13   private String placeholderList(Column[] columns)
14 }
```

- This class violates SRP becuase there are multiple axes of change, e.g., updating an exising statement type (like create) or adding new kinds of statements.
- Extension with new SQL query types requires modifying this class.

Georgia Tech

OCP Refactoring

Abstract base class that doesn't change:

```
1  public abstract class Sql {
2    public Sql(String table, Column[] columns)
3    public abstract String generate();
4  }
```

Extended by adding new subclasses without touching other classes:

```
1  public class CreateSql extends Sql {
2    public CreateSql(String table, Column[] columns)
3    @Override public String generate()
4  }
5  public class SelectSql extends Sql {
6    public SelectSql(String table, Column[] columns)
7    @Override public String generate()
8  }
```

This is high cohesion, low coupling, and reuse of the interface declared in the base class.

Georgia Tech

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their supertypes.

Most important principle in object-oriented design

Georgia Tech

LSP Counterexample

A suprising counter-example:

```
1  public class Rectangle {
2    public void setWidth(double w) { ... }
3    public void setHeight(double h) { ... }
4  }
5  public class Square extends Rectangle {
6    public void setWidth(double w) {
7      super.setWidth(w);
8      super.setHeight(w);
9    }
10   public void setHeight(double h) {
11     super.setWidth(h);
12     super.setHeight(h);
13   }
14 }
```

- We know from math class that a square "is a" rectangle.
- The overridden `setWidth` and `setHeight` methods in `Square` enforce the class invariant of `Square`, namely, that `width == height`.

LSP Violation

Consider this client of `Rectangle`:

```
1  public void g(Rectangle r) {
2    r.setWidth(5);
3    r.setHeight(4);
4    assert r.area() == 20;
5  }
```

- Client (author of `g`) assumes width and height are independent in `r` because `r` is a `Rectangle`.
- If the `r` passed to `g` is actually an instance of `Square`, what will be the value of `r.area()`?

The Object-oriented `is-a` relationship is about behavior. `Square`'s `setWidth` and `setHeight` methods don't behave the way a `Rectangle`'s `setWidth` and `setHeight` methods are expected to behave, so a `Square` doesn't fit the object-oriented *is-a* `Rectangle` definition. Let's make this more formal ...

Georgia Tech

Conforming to LSP: Design by Contract

Require no more, promise no less.

Author of a class specifies the behavior of each method in terms of preconditions and postconditions. Subclasses must follow two rules:

- Preconditions of overriden methods must be equal to or weaker than those of the superclass (enforces or assumes no more than the constraints of the superclass method).
- Postconditions of overriden methods must be equal to or greater than those of the superclass (enforces all of the constraints of the superclass method and possibly more).

In the Rectangle-Square case the postcondition of `Rectangle`'s `setWidth` method:

```
1 assert((rectangle.w == w) && (rectangle.height == old.height))
```

cannot be satisfied by `Square`, which tells us that a `Square` doesn't satisfy the object-oriented *is-a* relationship to `Rectangle`.

LSP Conforming 2D Shapes

```
1  public interface 2dShape {
2      double area();
3  }
4  public class Rectangle implements 2dShape {
5      public void setWidth(double w) { ... }
6      public void setHeight(double h) { ... }
7      public double area() {
8          return width * height;
9      }
10 }
11 public class Square implements 2dShape {
12     public void setSide(double w) { ... }
13     public double area() {
14         return side * side;
15     }
16 }
```

Georgia Tech

Interface Segregation Principle

Clients should not be forced to depend on methods they don't use.

Break up fat interfaces into a set of smaller interfaces. Each client depends on the small interface it needs, and none of the others.

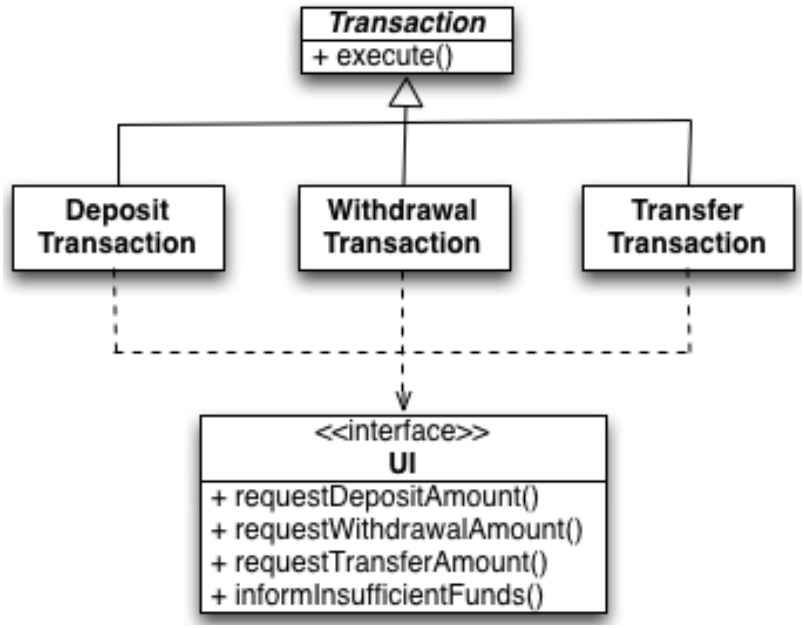

Figure 1: Fat UI Interface

Additional UI methods in UI require recompilation of all the transaction classes, even the ones that don't use the new methods.

Georgia Tech

ISP Refactoring

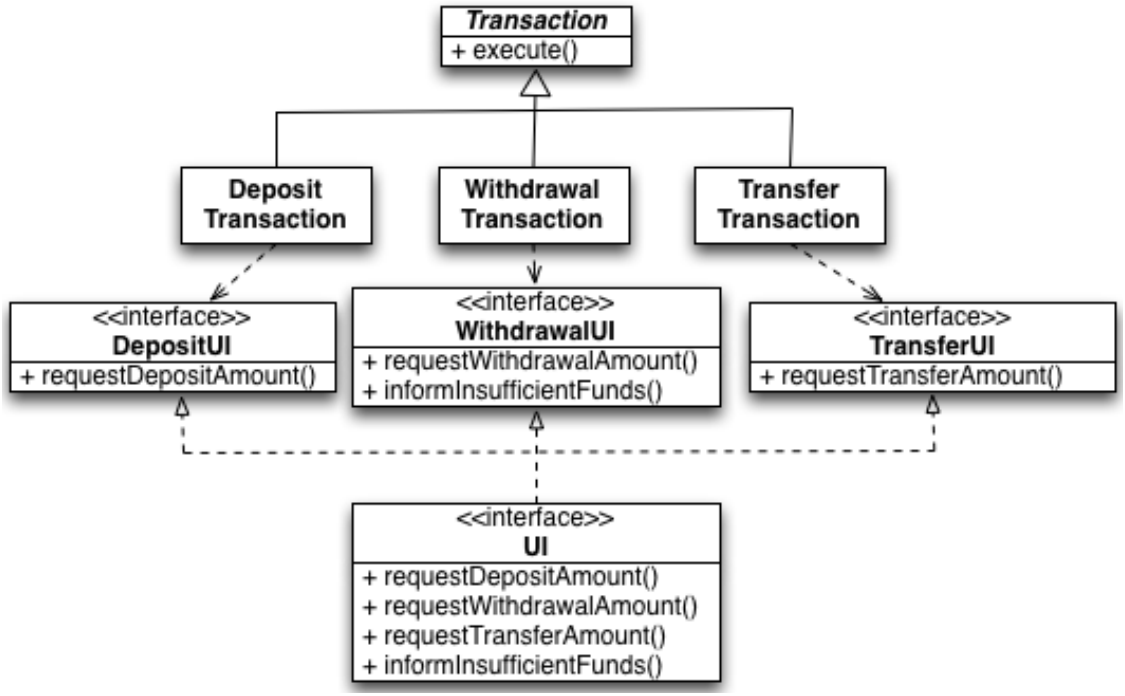

Figure 2: Segregated UI Interfaces

- Each transaction gets its own UI interface.
- Adding transactions doesn't require touching or recompiling other transactions or UIs.

Georgia Tech

Dependency Inversion Principle

a. High-level modules should not depend on low-level modules. Both should depend on abstractions.

This basically means program to an interface, not a particular implementation of the interface.

Georgia Tech

Dependency Inversion Principle

a. High-level modules should not depend on low-level modules. Both should depend on abstractions.

b. Abstractions should not depend on details. Details should depend on abstractions.

This basically means program to an interface, not a particular implementation of the interface.

Georgia Tech

DIP Counterexample[^1]

```
1  public class RealBillingService {
2    public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
3      PaypalCreditCardProcessor processor = new
           PaypalCreditCardProcessor();
4      // Card charging code ...
5    }
6  }
```

- Dependence on particular implementation of credit card processor

`new` is a code smell.

[^1] https://github.com/google/guice/wiki/Motivation

Georgia Tech

DIP Refactoring

```
1  public interface CreditCardProcessor { ... }
2
3  public class RealBillingService {
4    private final CreditCardProcessor processor;
5
6    public RealBillingService(CreditCardProcessor processor) {
7      this.processor = processor;
8    }
9
10   public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
11     // Credit card charging code ...
12   }
13 }
```

- Now `RealBillingService` depends on the `CreditCardProcessor` interface, not any particular implementation

Georgia Tech

Dependency Injection

```
1  public interface CreditCardProcessor { ... }
2
3  public class RealBillingService {
4    private final CreditCardProcessor processor;
5
6    public RealBillingService(CreditCardProcessor processor) {
7      this.processor = processor;
8    }
9
10   public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
         ... }
11 }
```

Note that we've eliminated `new` by passing an instance of `CreditCardPricessor` in the constructor

- This now satisfies the OCP because we can extend `RealBillingService` to work with additional `CreditCardProcessor`s without modifying `RealBillingService`
- Wiring a class to its concrete dependencies external to the class is known as *dependency injection* and it gets much fancier than the manual approach shown here

Documenting Designs – Unified Modeling Language

A standardized diagrammatic language for communicating OO designs in a language-independent way. Very rich, but for now focus on:

- use cases,
- domain model (classes and associations),
- packages, and
- sequence digrams.

Georgia Tech

UML Use Cases

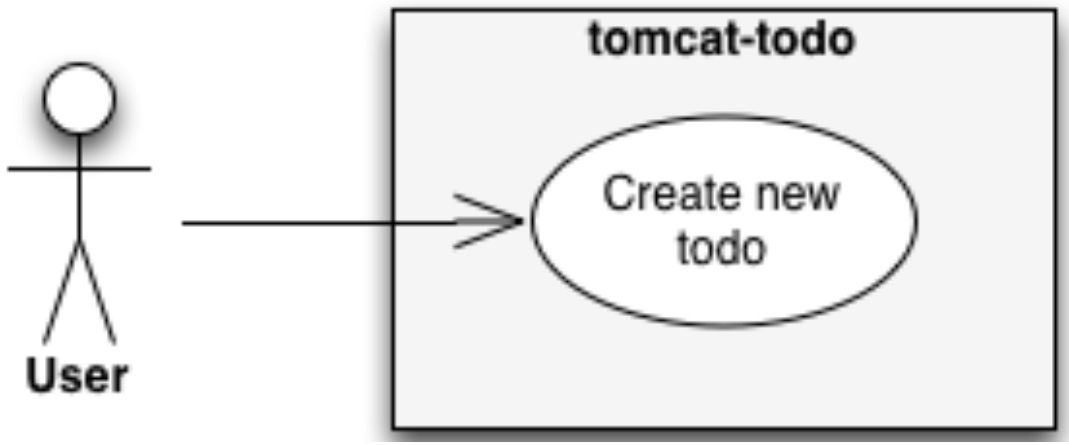

Figure 3: UML Use Case

A use case describes some user's interaction with the system. Most use cases contain:

- an actor, here simply "User,"
- a use case, here "Create new todo," and
- (optionally) a system boundary, here "tomcat-todo."

Georgia Tech

UML Class Diagrams

Figure 4: UML Class Diagram

Class diagrams contain

- a class name, here "Todo,"
- instance variables, here "title" and "task". Note that types are given after names, as in ": String". The "-" means private.
- methods. The "+" means public. ("#" means protected, but we have no protected members in this example.)

Georgia Tech

UML Associations

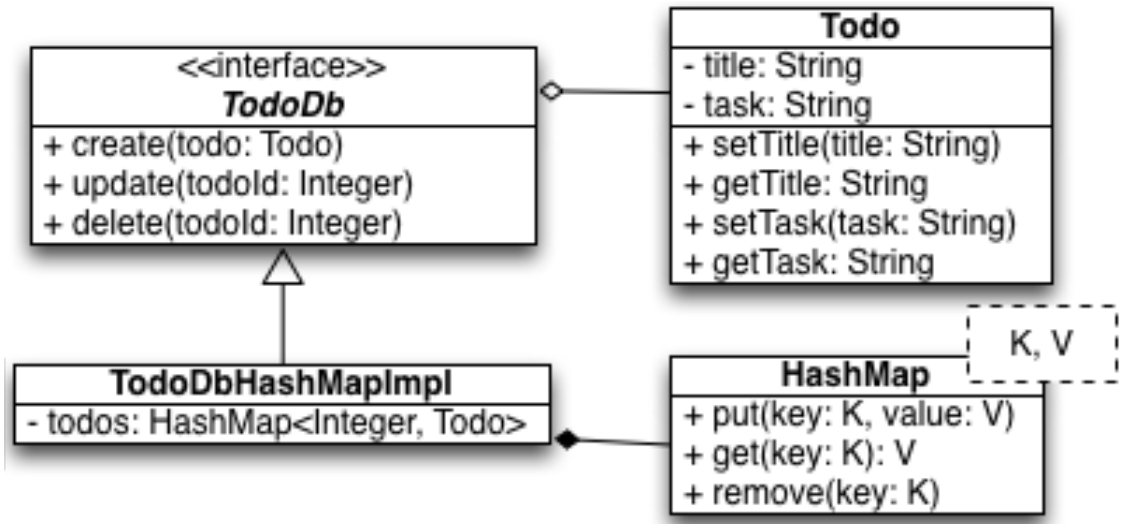

Figure 5: UML Associations

- /TodoDb/ is italicized, meaning it is abstract. The "<>" further means that it is an interface.
- `TodoDbHashMapImpl` is a subtype of `TodoDb`.
- `TodoDbHashMapImpl` is composed of a `HashMap<K,V>`.
- `HashMap` is a paramterized type with type parameters `K` and `V`.
- `TodoDb` aggregates `Todo` objects.

Georgia Tech

UML Packages

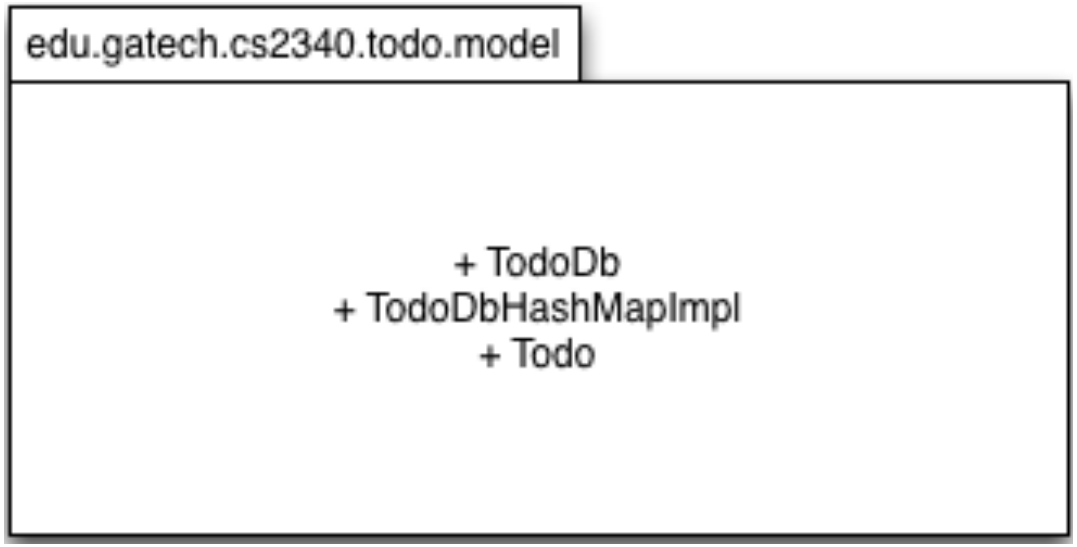

Figure 6: UML Package Diagram

- The tab shows the package name.
- The main box lists classes and interfaces using the same +, -, and # visibility modifiers used for members in class diagrams.
- An alternative form is to simply put the package name in the main box and not list the members of the package.

Georgia Tech

UML Sequence Diagrams

```
1  class TodoDbTreeMapImpl {
2    TreeMap<Integer,Todo>
          todos;
3
4    Integer create(Todo todo) {
5        Integer newId =
              todos.size();
6        todos.put(newId, todo);
7        return newId;
8    }
9  }
```

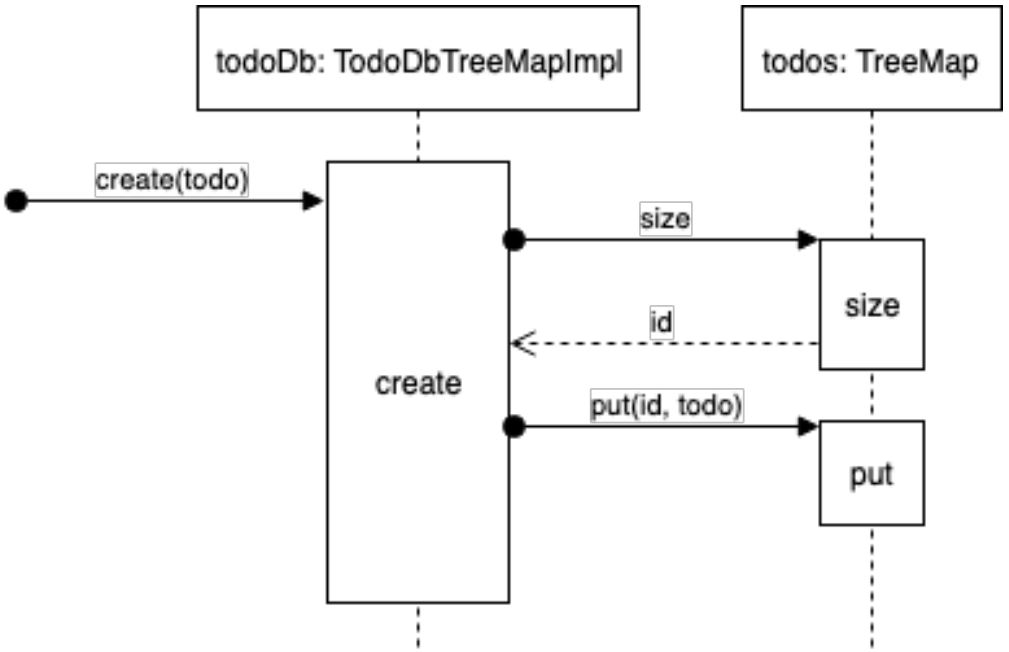

Figure 7: UML Sequence Diagram

- The top rectangles represent objects (instances of types/classes).
- Time progresses vertically downward.
- The dashed lines represent object lifetimes.
- The narrow vertical boxes represent operations of the objects (here, only create on the todoDb object).
- Arrows are "messages" or method calls and returns

Conclusions

- Design is an art born of empirical science and intuitive experience
- Practical experience and formal studies of software systems have produced design principles and guidelines
- Design involves tradeoffs – balancing constraints means prioritizing

And a final word about design documentation: if Jack Reeves is right (and I think he is), then

- The code is the design, produced by the design team,
- The compiler and linker are the manufacturing team, and
- Compilers and linkers don't use design documents.
- Design documents are for other "designers" to help them understand the code

Punch line: design documents are like comments – they make up for lack of expressivity in the code. They're a necessary evil at best, not the "point" of design.

Georgia Tech